Go 标准库
中文参考

wizardforcel

Published
with GitBook

目錄

Go 标准库 中文参考

> 来源：Go语言标准库文档中文版

库

子目录

标准库

名称	摘要
archive	
tar	tar包实现了tar格式压缩文件的存取.
zip	zip包提供了zip档案文件的读写服务.
bufio	bufio 包实现了带缓存的I/O操作.
builtin	builtin 包为Go的预声明标识符提供了文档.
bytes	bytes包实现了操作[]byte的常用函数.
compress	
bzip2	bzip2包实现bzip2的解压缩.
flate	flate包实现了deflate压缩数据格式，参见RFC 1951.
gzip	gzip包实现了gzip格式压缩文件的读写，参见RFC 1952.
lzw	lzw包实现了Lempel-Ziv-Welch数据压缩格式，这是一种T. A. Welch在“A Technique for High-Performance Data Compression”一文（Computer, 17(6) (June 1984), pp 8-19）提出的一种压缩格式.
zlib	zlib包实现了对zlib格式压缩数据的读写，参见RFC 1950.
container	
heap	heap包提供了对任意类型（实现了heap.Interface接口）的堆操作.
list	list包实现了双向链表.

ring	ring实现了环形链表的操作.
crypto	crypto包搜集了常用的密码（算法）常量.
aes	aes包实现了AES加密算法，参见U.S. Federal Information Processing Standards Publication 197.
cipher	cipher包实现了多个标准的用于包装底层块加密算法的加密算法实现.
des	des包实现了DES标准和TDEA算法，参见U.S. Federal Information Processing Standards Publication 46-3.
dsa	dsa包实现FIPS 186-3定义的数字签名算法（Digital Signature Algorithm），即DSA算法.
ecdsa	ecdsa包实现了椭圆曲线数字签名算法，参见FIPS 186-3.
elliptic	elliptic包实现了几条覆盖素数有限域的标准椭圆曲线.
hmac	hmac包实现了U.S. Federal Information Processing Standards Publication 198规定的HMAC（加密哈希信息认证码）.
md5	md5包实现了MD5哈希算法，参见RFC 1321.
rand	rand包实现了用于加解密的更安全的随机数生成器.
rc4	rc4包实现了RC4加密算法，参见Bruce Schneier's Applied Cryptography.
rsa	rsa包实现了PKCS#1规定的RSA加密算法.
sha1	sha1包实现了SHA1哈希算法，参见RFC 3174.
sha256	sha256包实现了SHA224和SHA256哈希算法，参见FIPS 180-4.
sha512	sha512包实现了SHA384和SHA512哈希算法，参见FIPS 180-2.
subtle	Package subtle implements functions that are often useful in cryptographic code but require careful thought to use correctly.
tls	tls包实现了TLS 1.2，细节参见RFC 5246.
x509	x509包解析X.509编码的证书和密钥.
pkix	pkix包提供了共享的、低层次的结构体，用于ASN.1解析和X.509证书、CRL、OCSP的序列化.
database	
sql	sql 包提供了通用的SQL（或类SQL）数据库接口.
driver	driver包定义了应被数据库驱动实现的接口，这些接口会被sql包使用.
debug	

dwarf	Package dwarf provides access to DWARF debugging information loaded from executable files, as defined in the DWARF 2.0 Standard at http://dwarfstd.org/doc/dwarf-2.0.0.pdf
elf	Package elf implements access to ELF object files.
gosym	Package gosym implements access to the Go symbol and line number tables embedded in Go binaries generated by the gc compilers.
macho	Package macho implements access to Mach-O object files.
pe	Package pe implements access to PE (Microsoft Windows Portable Executable) files.
plan9obj	Package plan9obj implements access to Plan 9 a.out object files.
encoding	encoding包定义了供其它包使用的可以将数据在字节水平和文本表示之间转换的接口.
ascii85	ascii85 包是对 ascii85 的数据编码的实现.
asn1	asn1包实现了DER编码的ASN.1数据结构的解析，参见ITU-T Rec X.690.
base32	base32包实现了RFC 4648规定的base32编码.
base64	base64实现了RFC 4648规定的base64编码.
binary	binary包实现了简单的数字与字节序列的转换以及变长值的编解码.
csv	csv读写逗号分隔值（csv）的文件.
gob	gob包管理gob流——在编码器（发送器）和解码器（接受器）之间交换的binary值.
hex	hex包实现了16进制字符表示的编解码.
json	json包实现了json对象的编解码，参见RFC 4627.
pem	pem包实现了PEM数据编码（源自保密增强邮件协议）.
xml	Package xml implements a simple XML 1.0 parser that understands XML name spaces.
errors	error 包实现了用于错误处理的函数.
expvar	expvar包提供了公共变量的标准接口，如服务的操作计数器.
flag	flag 包实现命令行标签解析.
fmt	fmt 包实现了格式化I/O函数，类似于C的 printf 和 scanf.
go	
	Package ast declares the types used to represent syntax trees

ast	for Go packages.
build	Package build gathers information about Go packages.
doc	Package doc extracts source code documentation from a Go AST.
format	Package format implements standard formatting of Go source.
parser	Package parser implements a parser for Go source files.
printer	Package printer implements printing of AST nodes.
scanner	Package scanner implements a scanner for Go source text.
token	Package token defines constants representing the lexical tokens of the Go programming language and basic operations on tokens (printing, predicates).
hash	hash包提供hash函数的接口.
adler32	adler32包实现了Adler-32校验和算法，参见RFC 1950.
crc32	crc32包实现了32位循环冗余校验（CRC-32）的校验和算法.
crc64	Package crc64 implements the 64-bit cyclic redundancy check, or CRC-64, checksum.
fnv	fnv包实现了FNV-1和FNV-1a（非加密hash函数）.
html	html包提供了用于转义和解转义HTML文本的函数.
template	template包（html/template）实现了数据驱动的模板，用于生成可对抗代码注入的安全HTML输出.
image	image实现了基本的2D图片库.
color	color 包实现了基本的颜色库。
palette	palette包提供了标准的调色板.
draw	draw 包提供组装图片的方法.
gif	gif 包实现了GIF图片的解码.
jpeg	jpeg包实现了jpeg格式图像的编解码.
png	png 包实现了PNG图像的编码和解码.
index	
suffixarray	suffixarrayb包通过使用内存中的后缀树实现了对数级时间消耗的子字符串搜索.
io	io 包为I/O原语提供了基础的接口.
ioutil	ioutil 实现了一些I/O的工具函数。

log	log包实现了简单的日志服务.
syslog	syslog包提供一个简单的系统日志服务的接口.
math	math 包提供了基本常数和数学函数。
big	big 包实现了（大数的）高精度运算.
cmplx	cmplx 包为复数提供了基本的常量和数学函数.
rand	rand 包实现了伪随机数生成器.
mime	mime实现了MIME的部分规定.
multipart	multipart实现了MIME的multipart解析，参见RFC 2046.
net	net包提供了可移植的网络I/O接口，包括TCP/IP、UDP、域名解析和Unix域socket.
http	http包提供了HTTP客户端和服务端的实现.
cgi	cgi 包实现了RFC3875协议描述的CGI（公共网关接口）.
cookiejar	cookiejar包实现了保管在内存中的符合RFC 6265标准的http.CookieJar接口.
fcgi	fcgi 包实现了FastCGI协议.
httptest	httptest 包提供HTTP测试的单元工具.
httputil	httputil包提供了HTTP公用函数，是对net/http包的更常见函数的补充.
pprof	pprof 包通过提供HTTP服务返回runtime的统计数据，这个数据是以pprof可视化工具规定的返回格式返回的.
mail	mail 包实现了解析邮件消息的功能.
rpc	rpc 包提供了一个方法来通过网络或者其他的I/O连接进入对象的外部方法.
jsonrpc	jsonrpc 包使用了rpc的包实现了一个JSON-RPC的客户端解码器和服务端的解码器.
smtp	smtp包实现了简单邮件传输协议（SMTP），参见RFC 5321.
textproto	textproto实现了对基于文本的请求/回复协议的一般性支持，包括HTTP、NNTP和SMTP.
url	url包解析URL并实现了查询的逸码，参见RFC 3986.
os	os包提供了操作系统函数的不依赖平台的接口.
exec	exec包执行外部命令.
signal	signal包实现了对输入信号的访问.

user	user包允许通过名称或ID查询用户帐户.
path	path实现了对斜杠分隔的路径的实用操作函数.
filepath	filepath包实现了兼容各操作系统的文件路径的实用操作函数.
reflect	reflect包实现了运行时反射，允许程序操作任意类型的对象.
regexp	regexp包实现了正则表达式搜索.
syntax	Package syntax parses regular expressions into parse trees and compiles parse trees into programs.
runtime	TODO(osc): 需更新 runtime 包含与Go的运行时系统进行交互的操作，例如用于控制Go程的函数.
cgo	cgo 包含有 cgo 工具生成的代码的运行时支持.
debug	debug 包含有程序在运行时调试其自身的功能.
pprof	pprof 包按照可视化工具 pprof 所要求的格式写出运行时分析数据.
race	race 包实现了数据竞争检测逻辑.
sort	sort 包为切片及用户定义的集合的排序操作提供了原语.
strconv	strconv包实现了基本数据类型和其字符串表示的相互转换.
strings	strings包实现了用于操作字符的简单函数.
sync	sync 包提供了互斥锁这类的基本的同步原语.
atomic	atomic 包提供了底层的原子性内存原语，这对于同步算法的实现很有用.
syscall	Package syscall contains an interface to the low-level operating system primitives.
testing	Package testing provides support for automated testing of Go packages.
iotest	Package iotest implements Readers and Writers useful mainly for testing.
quick	Package quick implements utility functions to help with black box testing.
text	
scanner	scanner包提供对utf-8文本的token扫描服务.
tabwriter	tabwriter包实现了写入过滤器（tabwriter.Writer），可以将输入的缩进修正为正确的对齐文本.
template	template包实现了数据驱动的用于生成文本输出的模板.

parse	text/template and html/template.
time	time包提供了时间的显示和测量用的函数.
unicode	unicode 包提供了一些测试Unicode码点属性的数据和函数.
utf16	utf16 包实现了对UTF-16序列的编码和解码。
utf8	utf8 包实现了支持UTF-8文本编码的函数和常量.
unsafe	unsafe 包含有关于Go程序类型安全的所有操作.

其它包

子代码库

这些包是 Go 项目的一部分，但并未在主源码树中。它们在比 Go 核心库更加宽松的兼容性需求下开发。 可通过“go get”安装它们，子代码库的文档和源码可通过相应的链接访问

- crypto — 附加的加密包。
- image — 附加的图像包。
- net — 附加的网络包。
- sys — 系统调用包。
- text — 文本处理包。
- tools — godoc、vet、cover 及其它工具。
- exp — 实验性代码（可能不经警告就更改，请小心对待）。

社区

这些服务可帮你寻找社区提供的开源包。

- GoDoc - 包索引与搜索引擎。
- Go 搜索 - 代码搜索引擎。
- Go 维基上的项目 - Go 项目策划列表

package achive

package tar

```
import "archive/tar"
```

tar包实现了tar格式压缩文件的存取。本包目标是覆盖大多数tar的变种，包括GNU和BSD生成的tar文件。

参见：

```
http://www.freebsd.org/cgi/man.cgi?query=tar&sektion=5
http://www.gnu.org/software/tar/manual/html_node/Standard.html
http://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html
```

Example

```
// Create a buffer to write our archive to.
buf := new(bytes.Buffer)
// Create a new tar archive.
tw := tar.NewWriter(buf)
// Add some files to the archive.
var files = []struct {
    Name, Body string
}{
    {"readme.txt", "This archive contains some text files."},
    {"gopher.txt", "Gopher names:\nGeorge\nGeoffrey\nGonzo"},
    {"todo.txt", "Get animal handling licence."},
}
for _, file := range files {
    hdr := &tar.Header{
        Name: file.Name,
        Size: int64(len(file.Body)),
    }
    if err := tw.WriteHeader(hdr); err != nil {
        log.Fatalln(err)
    }
    if _, err := tw.Write([]byte(file.Body)); err != nil {
        log.Fatalln(err)
    }
}
// Make sure to check the error on Close.
if err := tw.Close(); err != nil {
    log.Fatalln(err)
}
// Open the tar archive for reading.
r := bytes.NewReader(buf.Bytes())
tr := tar.NewReader(r)
// Iterate through the files in the archive.
for {
    hdr, err := tr.Next()
    if err == io.EOF {
        // end of tar archive
        break
    }
    if err != nil {
        log.Fatalln(err)
    }
    fmt.Printf("Contents of %s:\n", hdr.Name)
    if _, err := io.Copy(os.Stdout, tr); err != nil {
        log.Fatalln(err)
    }
    fmt.Println()
}
```

Output:

```
Contents of readme.txt:
This archive contains some text files.
Contents of gopher.txt:
Gopher names:
George
Geoffrey
Gonzo
Contents of todo.txt:
Get animal handling licence.
```

Index

Examples

Constants

```
const (
    // 类型
    TypeReg           = '0'    // 普通文件
    TypeRegA          = '\x00' // 普通文件
    TypeLink          = '1'    // 硬链接
    TypeSymlink       = '2'    // 符号链接
    TypeChar          = '3'    // 字符设备节点
    TypeBlock         = '4'    // 块设备节点
    TypeDir           = '5'    // 目录
    TypeFifo          = '6'    // 先进先出队列节点
    TypeCont          = '7'    // 保留位
    TypeXHeader       = 'x'    // 扩展头
    TypeXGlobalHeader = 'g'    // 全局扩展头
    TypeGNULongName   = 'L'    // 下一个文件记录有个长名字
    TypeGNULongLink   = 'K'    // 下一个文件记录指向一个具有长名字的文件
    TypeGNUSparse     = 'S'    // 稀疏文件
)
```

Variables

```
var (
    ErrWriteTooLong    = errors.New("archive/tar: write too long")
    ErrFieldTooLong    = errors.New("archive/tar: header field too
    ErrWriteAfterClose = errors.New("archive/tar: write after close
)
```

```
var (
    ErrHeader = errors.New("archive/tar: invalid tar header")
)
```

type Header

```
type Header struct {
    Name       string    // 记录头域的文件名
    Mode       int64     // 权限和模式位
    Uid        int       // 所有者的用户ID
    Gid        int       // 所有者的组ID
    Size       int64     // 字节数（长度）
    ModTime    time.Time // 修改时间
    Typeflag   byte      // 记录头的类型
    Linkname   string    // 链接的目标名
    Uname      string    // 所有者的用户名
    Gname      string    // 所有者的组名
    Devmajor   int64     // 字符设备或块设备的major number
    Devminor   int64     // 字符设备或块设备的minor number
    AccessTime time.Time // 访问时间
    ChangeTime time.Time // 状态改变时间
    Xattrs     map[string]string
}
```

Header代表tar档案文件里的单个头。Header类型的某些字段可能未使用。

func FileInfoHeader

```
func FileInfoHeader(fi os.FileInfo, link string) (*Header, error)
```

FileInfoHeader返回一个根据fi填写了部分字段的Header。 如果fi描述一个符号链接，FileInfoHeader函数将link参数作为链接目标。如果fi描述一个目录，会在名字后面添加斜杠。因为os.FileInfo接口的Name方法只返回它描述的文件的无路径名，有可能需要将返回值的Name字段修改为文件的完整路径名。

func (*Header) FileInfo

```
func (h *Header) FileInfo() os.FileInfo
```

FileInfo返回该Header对应的文件信息。（os.FileInfo类型）

type Reader

```
type Reader struct {
    // 内含隐藏或非导出字段
}
```

Reader提供了对一个tar档案文件的顺序读取。一个tar档案文件包含一系列文件。Next方法返回档案中的下一个文件（包括第一个），返回值可以被视为io.Reader来获取文件的数据。

func NewReader

```
func NewReader(r io.Reader) *Reader
```

NewReader创建一个从r读取的Reader。

func (*Reader) Next

```
func (tr *Reader) Next() (*Header, error)
```

转入tar档案文件下一记录，它会返回下一记录的头域。

func (*Reader) Read

```
func (tr *Reader) Read(b []byte) (n int, err error)
```

从档案文件的当前记录读取数据，到达记录末端时返回(0, EOF)，直到调用Next方法转入下一记录。

type Writer

```
type Writer struct {
    // 内含隐藏或非导出字段
}
```

Writer类型提供了POSIX.1格式的tar档案文件的顺序写入。一个tar档案文件包含一系列文件。调用WriteHeader来写入一个新的文件，然后调用Write写入文件的数据，该记录写入的数据不能超过hdr.Size字节。

func NewWriter

```
func NewWriter(w io.Writer) *Writer
```

NewWriter创建一个写入w的*Writer。

func (*Writer) WriteHeader

```
func (tw *Writer) WriteHeader(hdr *Header) error
```

WriteHeader写入hdr并准备接受文件内容。如果不是第一次调用本方法，会调用Flush。在Close之后调用本方法会返回ErrWriteAfterClose。

func (*Writer) Write

```
func (tw *Writer) Write(b []byte) (n int, err error)
```

Write向tar档案文件的当前记录中写入数据。如果写入的数据总数超出上一次调用WriteHeader的参数hdr.Size字节，返回ErrWriteTooLong错误。

func (*Writer) Flush

```
func (tw *Writer) Flush() error
```

Flush结束当前文件的写入。（可选的）

func (*Writer) Close

```
func (tw *Writer) Close() error
```

Close关闭tar档案文件，会将缓冲中未写入下层的io.Writer接口的数据刷新到下层。

package zip

```
import "archive/zip"
```

zip包提供了zip档案文件的读写服务。参见 http://www.pkware.com/documents/casestudies/APPNOTE.TXT

本包不支持跨硬盘的压缩。

关于ZIP64：

为了向下兼容，FileHeader同时拥有32位和64位的Size字段。64位字段总是包含正确的值，对普通格式的档案未见它们的值是相同的。对zip64格式的档案文件32位字段将是0xffffffff，必须使用64位字段。

Index

Examples

Constants

```
const (
    Store   uint16 = 0
    Deflate uint16 = 8
)
```

预定义压缩算法。

Variables

```
var (
    ErrFormat    = errors.New("zip: not a valid zip file")
    ErrAlgorithm = errors.New("zip: unsupported compression algorit
    ErrChecksum  = errors.New("zip: checksum error")
)
```

type Compressor

```
type Compressor func(io.Writer) (io.WriteCloser, error)
```

Compressor函数类型会返回一个io.WriteCloser，该接口会将数据压缩后写入提供的接口。关闭时，应将缓冲中的数据刷新到下层接口中。

type Decompressor

```
type Decompressor func(io.Reader) io.ReadCloser
```

Decompressor函数类型会返回一个io.ReadCloser， 该接口的Read方法会将读取自提供的接口的数据提前解压缩。程序员有责任在读取结束时关闭该io.ReadCloser。

func RegisterCompressor

```
func RegisterCompressor(method uint16, comp Compressor)
```

RegisterCompressor使用指定的方法ID注册一个Compressor类型函数。常用的方法Store和Deflate是内建的。

func RegisterDecompressor

```
func RegisterDecompressor(method uint16, d Decompressor)
```

RegisterDecompressor使用指定的方法ID注册一个Decompressor类型函数。

type FileHeader

```
type FileHeader struct {
    // Name是文件名，它必须是相对路径，不能以设备或斜杠开始，只接受'/'作为路径
    Name string
    CreatorVersion     uint16
    ReaderVersion      uint16
    Flags              uint16
    Method             uint16
    ModifiedTime       uint16 // MS-DOS时间
    ModifiedDate       uint16 // MS-DOS日期
    CRC32              uint32
    CompressedSize     uint32 // 已弃用；请使用CompressedSize64
    UncompressedSize   uint32 // 已弃用；请使用UncompressedSize64
    CompressedSize64   uint64
    UncompressedSize64 uint64
    Extra              []byte
    ExternalAttrs      uint32 // 其含义依赖于CreatorVersion
    Comment            string
}
```

FileHeader描述zip文件中的一个文件。参见zip的定义获取细节。

func FileInfoHeader

```
func FileInfoHeader(fi os.FileInfo) (*FileHeader, error)
```

FileInfoHeader返回一个根据fi填写了部分字段的Header。因为os.FileInfo接口的Name方法只返回它描述的文件的无路径名，有可能需要将返回值的Name字段修改为文件的完整路径名。

func (*FileHeader) FileInfo

```
func (h *FileHeader) FileInfo() os.FileInfo
```

FileInfo返回一个根据h的信息生成的os.FileInfo。

func (*FileHeader) Mode

```
func (h *FileHeader) Mode() (mode os.FileMode)
```

Mode返回h的权限和模式位。

func (*FileHeader) SetMode

```
func (h *FileHeader) SetMode(mode os.FileMode)
```

SetMode修改h的权限和模式位。

func (*FileHeader) ModTime

```
func (h *FileHeader) ModTime() time.Time
```

返回最近一次修改的UTC时间。（精度2s）

func (*FileHeader) SetModTime

```
func (h *FileHeader) SetModTime(t time.Time)
```

将ModifiedTime和ModifiedDate字段设置为给定的UTC时间。（精度2s）

type File

```
type File struct {
    FileHeader
    // 内含隐藏或非导出字段
}
```

func (*File) DataOffset

```
func (f *File) DataOffset() (offset int64, err error)
```

DataOffset返回文件的可能存在的压缩数据相对于zip文件起始的偏移量。大多数调用者应使用Open代替，该方法会主动解压缩数据并验证校验和。

func (*File) Open

```
func (f *File) Open() (rc io.ReadCloser, err error)
```

Open方法返回一个io.ReadCloser接口，提供读取文件内容的方法。可以同时读取多个文件。

type Reader

```
type Reader struct {
    File    []*File
    Comment string
    // 内含隐藏或非导出字段
}
```

Example

```
// Open a zip archive for reading.
r, err := zip.OpenReader("testdata/readme.zip")
if err != nil {
    log.Fatal(err)
}
defer r.Close()
// Iterate through the files in the archive,
// printing some of their contents.
for _, f := range r.File {
    fmt.Printf("Contents of %s:\n", f.Name)
    rc, err := f.Open()
    if err != nil {
        log.Fatal(err)
    }
    _, err = io.CopyN(os.Stdout, rc, 68)
    if err != nil {
        log.Fatal(err)
    }
    rc.Close()
    fmt.Println()
}
```

Output:

```
Contents of README:
This is the source code repository for the Go programming language
```

func NewReader

```
func NewReader(r io.ReaderAt, size int64) (*Reader, error)
```

NewReader返回一个从r读取数据的*Reader，r被假设其大小为size字节。

type ReadCloser

```
type ReadCloser struct {
    Reader
    // 内含隐藏或非导出字段
}
```

func OpenReader

```
func OpenReader(name string) (*ReadCloser, error)
```

OpenReader会打开name指定的zip文件并返回一个*ReadCloser。

func (*ReadCloser) Close

```
func (rc *ReadCloser) Close() error
```

Close关闭zip文件，使它不能用于I/O。

type Writer

```
type Writer struct {
    // 内含隐藏或非导出字段
}
```

Writer类型实现了zip文件的写入器。

Example

```
// Create a buffer to write our archive to.
buf := new(bytes.Buffer)
// Create a new zip archive.
w := zip.NewWriter(buf)
// Add some files to the archive.
var files = []struct {
    Name, Body string
}{
    {"readme.txt", "This archive contains some text files."},
    {"gopher.txt", "Gopher names:\nGeorge\nGeoffrey\nGonzo"},
    {"todo.txt", "Get animal handling licence.\nWrite more examples
}
for _, file := range files {
    f, err := w.Create(file.Name)
    if err != nil {
        log.Fatal(err)
    }
    _, err = f.Write([]byte(file.Body))
    if err != nil {
        log.Fatal(err)
    }
}
// Make sure to check the error on Close.
err := w.Close()
if err != nil {
    log.Fatal(err)
}
```

func NewWriter

```
func NewWriter(w io.Writer) *Writer
```

NewWriter创建并返回一个将zip文件写入w的*Writer。

func (*Writer) CreateHeader

```
func (w *Writer) CreateHeader(fh *FileHeader) (io.Writer, error)
```

使用给出的*FileHeader来作为文件的元数据添加一个文件进zip文件。本方法返回一个io.Writer接口（用于写入新添加文件的内容）。新增文件的内容必须在下一次调用CreateHeader、Create或Close方法之前全部写入。

func (*Writer) Create

```
func (w *Writer) Create(name string) (io.Writer, error)
```

使用给出的文件名添加一个文件进zip文件。本方法返回一个io.Writer接口（用于写入新添加文件的内容）。文件名必须是相对路径，不能以设备或斜杠开始，只接受'/'作为路径分隔。新增文件的内容必须在下一次调用CreateHeader、Create或Close方法之前全部写入。

func (*Writer) Close

```
func (w *Writer) Close() error
```

Close方法通过写入中央目录关闭该*Writer。本方法不会也没办法关闭下层的io.Writer接口。

package bufio

```
import "bufio"
```

bufio包实现了有缓冲的I/O。它包装一个io.Reader或io.Writer接口对象，创建另一个也实现了该接口，且同时还提供了缓冲和一些文本I/O的帮助函数的对象。

Index

Examples

Constants

```
const (
    // 用于缓冲一个token，实际需要的最大token尺寸可能小一些，例如缓冲中需要保
    MaxScanTokenSize = 64 * 1024
)
```

Variables

```
var (
    ErrInvalidUnreadByte = errors.New("bufio: invalid use of Unreac
    ErrInvalidUnreadRune = errors.New("bufio: invalid use of Unreac
    ErrBufferFull        = errors.New("bufio: buffer full")
    ErrNegativeCount     = errors.New("bufio: negative count")
)
```

```
var (
    ErrTooLong         = errors.New("bufio.Scanner: token too long'
    ErrNegativeAdvance = errors.New("bufio.Scanner: SplitFunc retur
    ErrAdvanceTooFar   = errors.New("bufio.Scanner: SplitFunc retur
)
```

会被Scanner类型返回的错误。

type Reader

```
type Reader struct {
    // 内含隐藏或非导出字段
}
```

Reader实现了给一个io.Reader接口对象附加缓冲。

func NewReader

```
func NewReader(rd io.Reader) *Reader
```

NewReader创建一个具有默认大小缓冲、从r读取的*Reader。

func NewReaderSize

```
func NewReaderSize(rd io.Reader, size int) *Reader
```

NewReaderSize创建一个具有最少有size尺寸的缓冲、从r读取的*Reader。如果参数r已经是一个具有足够大缓冲的* Reader类型值，会返回r。

func (*Reader) Reset

```
func (b *Reader) Reset(r io.Reader)
```

Reset丢弃缓冲中的数据，清除任何错误，将b重设为其下层从r读取数据。

func (*Reader) Buffered

```
func (b *Reader) Buffered() int
```

Buffered返回缓冲中现有的可读取的字节数。

func (*Reader) Peek

```
func (b *Reader) Peek(n int) ([]byte, error)
```

Peek返回输入流的下n个字节，而不会移动读取位置。返回的[]byte只在下一次调用读取操作前合法。如果Peek返回的切片长度比n小，它也会返会一个错误说明原因。如果n比缓冲尺寸还大，返回的错误将是ErrBufferFull。

func (*Reader) Read

```
func (b *Reader) Read(p []byte) (n int, err error)
```

Read读取数据写入p。本方法返回写入p的字节数。本方法一次调用最多会调用下层Reader接口一次Read方法，因此返回值n可能小于len(p)。读取到达结尾时，返回值n将为0而err将为io.EOF。

func (*Reader) ReadByte

```
func (b *Reader) ReadByte() (c byte, err error)
```

ReadByte读取并返回一个字节。如果没有可用的数据，会返回错误。

func (*Reader) UnreadByte

```
func (b *Reader) UnreadByte() error
```

UnreadByte吐出最近一次读取操作读取的最后一个字节。（只能吐出最后一个，多次调用会出问题）

func (*Reader) ReadRune

```
func (b *Reader) ReadRune() (r rune, size int, err error)
```

ReadRune读取一个utf-8编码的unicode码值，返回该码值、其编码长度和可能的错误。如果utf-8编码非法，读取位置只移动1字节，返回U+FFFD，返回值size为1而err为nil。如果没有可用的数据，会返回错误。

func (*Reader) UnreadRune

```
func (b *Reader) UnreadRune() error
```

UnreadRune吐出最近一次ReadRune调用读取的unicode码值。如果最近一次读取不是调用的ReadRune，会返回错误。（从这点看，UnreadRune比UnreadByte严格很多）

func (*Reader) ReadLine

```
func (b *Reader) ReadLine() (line []byte, isPrefix bool, err error)
```

ReadLine是一个低水平的行数据读取原语。大多数调用者应使用ReadBytes('\n')或ReadString('\n')代替，或者使用Scanner。

ReadLine尝试返回一行数据，不包括行尾标志的字节。如果行太长超过了缓冲，返回值isPrefix会被设为true，并返回行的前面一部分。该行剩下的部分将在之后的调用中返回。返回值isPrefix会在返回该行最后一个片段时才设为false。返回切片是缓冲的子切片，只在下一次读取操作之前有效。ReadLine要么返回一个非nil的line，要么返回一个非nil的err，两个返回值至少一个非nil。

返回的文本不包含行尾的标志字节（"\r\n"或"\n"）。如果输入流结束时没有行尾标志字节，方法不会出错，也不会指出这一情况。在调用ReadLine之后调用UnreadByte会总是吐出最后一个读取的字节（很可能是该行的行尾标志字节），即使该字节不是ReadLine返回值的一部分。

func (*Reader) ReadSlice

```
func (b *Reader) ReadSlice(delim byte) (line []byte, err error)
```

ReadSlice读取直到第一次遇到delim字节，返回缓冲里的包含已读取的数据和delim字节的切片。该返回值只在下一次读取操作之前合法。如果ReadSlice放在在读取到delim之前遇到了错误，它会返回在错误之前读取的数据在缓冲中的切片以及该错误（一般是io.EOF）。如果在读取到delim之前缓冲就被写满了，ReadSlice失败并返回ErrBufferFull。因为ReadSlice的返回值会被下一次I/O操作重写，调用者应尽量使用ReadBytes或ReadString替代本法功法。当且仅当ReadBytes方法返回的切片不以delim结尾时，会返回一个非nil的错误。

func (*Reader) ReadBytes

```
func (b *Reader) ReadBytes(delim byte) (line []byte, err error)
```

ReadBytes读取直到第一次遇到delim字节，返回一个包含已读取的数据和delim字节的切片。如果ReadBytes方法在读取到delim之前遇到了错误，它会返回在错误之前读取的数据以及该错误（一般是io.EOF）。当且仅当ReadBytes方法返回的切片不以delim结尾时，会返回一个非nil的错误。

func (*Reader) ReadString

```
func (b *Reader) ReadString(delim byte) (line string, err error)
```

ReadString读取直到第一次遇到delim字节，返回一个包含已读取的数据和delim字节的字符串。如果ReadString方法在读取到delim之前遇到了错误，它会返回在错误之前读取的数据以及该错误（一般是io.EOF）。当且仅当ReadString方法返回的切片不以delim结尾时，会返回一个非nil的错误。

func (*Reader) WriteTo

```
func (b *Reader) WriteTo(w io.Writer) (n int64, err error)
```

WriteTo方法实现了io.WriterTo接口。

type Writer

```
type Writer struct {
    // 内含隐藏或非导出字段
}
```

Writer实现了为io.Writer接口对象提供缓冲。如果在向一个Writer类型值写入时遇到了错误，该对象将不再接受任何数据，且所有写操作都会返回该错误。在说有数据都写入后，调用者有义务调用Flush方法以保证所有的数据都交给了下层的io.Writer。

Example

```
w := bufio.NewWriter(os.Stdout)
fmt.Fprint(w, "Hello, ")
fmt.Fprint(w, "world!")
w.Flush() // Don't forget to flush!
```

Output:

```
Hello, world!
```

func NewWriter

```
func NewWriter(w io.Writer) *Writer
```

NewWriter创建一个具有默认大小缓冲、写入w的*Writer。

func NewWriterSize

```
func NewWriterSize(w io.Writer, size int) *Writer
```

NewWriterSize创建一个具有最少有size尺寸的缓冲、写入w的*Writer。如果参数w已经是一个具有足够大缓冲的*Writer类型值，会返回w。

func (*Writer) Reset

```
func (b *Writer) Reset(w io.Writer)
```

Reset丢弃缓冲中的数据，清除任何错误，将b重设为将其输出写入w。

func (*Writer) Buffered

```
func (b *Writer) Buffered() int
```

Buffered返回缓冲中已使用的字节数。

func (*Writer) Available

```
func (b *Writer) Available() int
```

Available返回缓冲中还有多少字节未使用。

func (*Writer) Write

```
func (b *Writer) Write(p []byte) (nn int, err error)
```

Write将p的内容写入缓冲。返回写入的字节数。如果返回值nn < len(p)，还会返回一个错误说明原因。

func (*Writer) WriteString

```
func (b *Writer) WriteString(s string) (int, error)
```

WriteString写入一个字符串。返回写入的字节数。如果返回值nn < len(s)，还会返回一个错误说明原因。

func (*Writer) WriteByte

```
func (b *Writer) WriteByte(c byte) error
```

WriteByte写入单个字节。

func (*Writer) WriteRune

```
func (b *Writer) WriteRune(r rune) (size int, err error)
```

WriteRune写入一个unicode码值（的utf-8编码），返回写入的字节数和可能的错误。

func (*Writer) Flush

```
func (b *Writer) Flush() error
```

Flush方法将缓冲中的数据写入下层的io.Writer接口。

func (*Writer) ReadFrom

```
func (b *Writer) ReadFrom(r io.Reader) (n int64, err error)
```

ReadFrom实现了io.ReaderFrom接口。

type ReadWriter

```
type ReadWriter struct {
    *Reader
    *Writer
}
```

ReadWriter类型保管了指向Reader和Writer类型的指针，（因此）实现了io.ReadWriter接口。

func NewReadWriter

```
func NewReadWriter(r *Reader, w *Writer) *ReadWriter
```

NewReadWriter申请创建一个新的、将读写操作分派给r和w 的ReadWriter。

type SplitFunc

```
type SplitFunc func(data []byte, atEOF bool) (advance int, token [
```

SplitFunc类型代表用于对输出作词法分析的分割函数。

参数data是尚未处理的数据的一个开始部分的切片，参数atEOF表示是否Reader接口不能提供更多的数据。返回值是解析位置前进的字节数，将要返回给调用者的token切片，以及可能遇到的错误。如果数据不足以（保证）生成一个完整的token，例如需要一整行数据但data里没有换行符，SplitFunc可以返回(0, nil, nil)来告诉Scanner读取更多的数据写入切片然后用从同一位置起始、长度更长的切片再试一次（调用SplitFunc类型函数）。

如果返回值err非nil，扫描将终止并将该错误返回给Scanner的调用者。

除非atEOF为真，永远不会使用空切片data调用SplitFunc类型函数。然而，如果atEOF为真，data却可能是非空的、且包含着未处理的文本。

func ScanBytes

```
func ScanBytes(data []byte, atEOF bool) (advance int, token []byte,
```

ScanBytes是用于Scanner类型的分割函数（符合SplitFunc），本函数会将每个字节作为一个token返回。

func ScanRunes

```
func ScanRunes(data []byte, atEOF bool) (advance int, token []byte,
```

ScanRunes是用于Scanner类型的分割函数（符合SplitFunc），本函数会将每个utf-8编码的unicode码值作为一个token返回。本函数返回的rune序列和range一个字符串的输出rune序列相同。错误的utf-8编码会翻译为U+FFFD = "\xef\xbf\xbd"，但只会消耗一个字节。调用者无法区分正确编码的rune和错误编码的rune。

func ScanWords

```
func ScanWords(data []byte, atEOF bool) (advance int, token []byte,
```

ScanRunes是用于Scanner类型的分割函数（符合SplitFunc），本函数会将空白（参见unicode.IsSpace）分隔的片段（去掉前后空白后）作为一个token返回。本函数永远不会返回空字符串。

func ScanLines

```
func ScanLines(data []byte, atEOF bool) (advance int, token []byte,
```

ScanRunes是用于Scanner类型的分割函数（符合SplitFunc），本函数会将每一行文本去掉末尾的换行标记作为一个token返回。返回的行可以是空字符串。换行标记为一个可选的回车后跟一个必选的换行符。最后一行即使没有换行符也会作为一个token返回。

type Scanner

```
type Scanner struct {
    // 内含隐藏或非导出字段
}
```

Scanner类型提供了方便的读取数据的接口，如从换行符分隔的文本里读取每一行。

成功调用的Scan方法会逐步提供文件的token，跳过token之间的字节。token由SplitFunc类型的分割函数指定；默认的分割函数会将输入分割为多个行，并去掉行尾的换行标志。本包预定义的分割函数可以将文件分割为行、字节、unicode码值、空白分隔的word。调用者可以定制自己的分割函数。

扫描会在抵达输入流结尾、遇到的第一个I/O错误、token过大不能保存进缓冲时，不可恢复的停止。当扫描停止后，当前读取位置可能会远在最后一个获得的token后面。需要更多对错误管理的控制或token很大，或必须从reader连续扫描的程序，应使用bufio.Reader代替。

Example (Custom)

```
// An artificial input source.
const input = "1234 5678 1234567901234567890"
scanner := bufio.NewScanner(strings.NewReader(input))
// Create a custom split function by wrapping the existing ScanWord
split := func(data []byte, atEOF bool) (advance int, token []byte,
    advance, token, err = bufio.ScanWords(data, atEOF)
    if err == nil && token != nil {
        _, err = strconv.ParseInt(string(token), 10, 32)
    }
    return
}
// Set the split function for the scanning operation.
scanner.Split(split)
// Validate the input
for scanner.Scan() {
    fmt.Printf("%s\n", scanner.Text())
}
if err := scanner.Err(); err != nil {
    fmt.Printf("Invalid input: %s", err)
}
```

Output:

```
1234
5678
Invalid input: strconv.ParseInt: parsing "1234567901234567890": val
```

Example (Lines)

```
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
    fmt.Println(scanner.Text()) // Println will add back the final
}
if err := scanner.Err(); err != nil {
    fmt.Fprintln(os.Stderr, "reading standard input:", err)
}
```

Example (Words)

```
// An artificial input source.
const input = "Now is the winter of our discontent,\nMade glorious
scanner := bufio.NewScanner(strings.NewReader(input))
// Set the split function for the scanning operation.
scanner.Split(bufio.ScanWords)
// Count the words.
count := 0
for scanner.Scan() {
    count++
}
if err := scanner.Err(); err != nil {
    fmt.Fprintln(os.Stderr, "reading input:", err)
}
fmt.Printf("%d\n", count)
```

Output:

```
15
```

func NewScanner

```
func NewScanner(r io.Reader) *Scanner
```

NewScanner创建并返回一个从r读取数据的Scanner，默认的分割函数是ScanLines。

func (*Scanner) Split

```
func (s *Scanner) Split(split SplitFunc)
```

Split设置该Scanner的分割函数。本方法必须在Scan之前调用。

func (*Scanner) Scan

```
func (s *Scanner) Scan() bool
```

Scan方法获取当前位置的token（该token可以通过Bytes或Text方法获得），并让Scanner的扫描位置移动到下一个token。当扫描因为抵达输入流结尾或者遇到错误而停止时，本方法会返回false。在Scan方法返回false后，Err方法将返回扫描时遇到的任何错误；除非是io.EOF，此时Err会返回nil。

func (*Scanner) Bytes

```
func (s *Scanner) Bytes() []byte
```

Bytes方法返回最近一次Scan调用生成的token。底层数组指向的数据可能会被下一次Scan的调用重写。

func (*Scanner) Text

```
func (s *Scanner) Text() string
```

Bytes方法返回最近一次Scan调用生成的token，会申请创建一个字符串保存token并返回该字符串。

func (*Scanner) Err

```
func (s *Scanner) Err() error
```

Err返回Scanner遇到的第一个非EOF的错误。

package builtin

```
import "builtin"
```

builtin 包为Go的预声明标识符提供了文档。此处列出的条目其实并不在builtin 包中，对它们的描述只是为了让 godoc 给该语言的特殊标识符提供文档。

Index

- func close(c chan<- Type)
- func panic(v interface{})
- func recover() interface{}
- func print(args ...Type)
- func println(args ...Type)

Constants

```
const (
    true  = 0 == 0 // 无类型布尔值
    false = 0 != 0 // 无类型布尔值
)
```

true 和false是两个无类型布尔值。

```
const iota = 0 // 无类型整数值
```

iota是一个预定义的标识符，代表顺序按行增加的无符号整数，每个const声明单元（被括号括起来）相互独立，分别从0开始。

type bool

```
type bool bool
```

布尔类型。

type byte

```
type byte byte
```

8位无符号整型，是uint8的别名，二者视为同一类型。

type rune

```
type rune rune
```

32位有符号整形，int32的别名，二者视为同一类型。

type int

```
type int int
```

至少32位的有符号整形，但和int32/rune并非同一类型。

type int8

```
type int8 int8
```

8位有符号整形，范围[-128, 127]。

type int16

```
type int16 int16
```

16位有符号整形，范围[-32768, 32767]。

type int32

```
type int32 int32
```

32位有符号整形，范围[-2147483648, 2147483647]。

type int64

```
type int64 int64
```

64位有符号整形，范围[-9223372036854775808, 9223372036854775807]。

type uint

```
type uint uint
```

至少32位的无符号整形，但和uint32不是同一类型。

type uint8

```
type uint8 uint8
```

8位无符号整型，范围[0, 255]。

type uint16

```
type uint16 uint16
```

16位无符号整型，范围[0, 65535]。

type uint32

```
type uint32 uint32
```

32位无符号整型，范围[0, 4294967295]。

type uint64

```
type uint64 uint64
```

64位无符号整型，范围[0, 18446744073709551615]。

type float32

```
type float32 float32
```

所有IEEE-754 32位浮点数的集合，12位有效数字。

type float64

```
type float64 float64
```

所有IEEE-754 64位浮点数的集合，16位有效数字。

type complex64

```
type complex64 complex64
```

具有float32 类型实部和虚部的复数类型。

type complex128

```
type complex128 complex128
```

具有float64 类型实部和虚部的复数类型。

type uintptr

```
type uintptr uintptr
```

可以保存任意指针的位模式的整数类型。

type string

```
type string string
```

8位byte序列构成的字符串，约定但不必须是utf-8编码的文本。字符串可以为空但不能是nil，其值不可变。

type error

```
type error interface {
    Error() string
}
```

内建error接口类型是约定用于表示错误信息，nil值表示无错误。

type Type

```
type Type int
```

在本文档中代表任意一个类型，但同一个声明里只代表同一个类型。

```
var nil Type // Type必须是指针、通道、函数、接口、映射或切片
```

nil是预定义的标识符，代表指针、通道、函数、接口、映射或切片的零值。

type Type1

```
type Type1 int
```

在本文档中代表任意一个类型，但同一个声明里只代表同一个类型，用于代表和Type不同的另一类型。

type IntegerType

```
type IntegerType int
```

在本文档中代表一个有符号或无符号的整数类型。

type FloatType

```
type FloatType float32
```

在本文档中代表一个浮点数类型。

type ComplexType

```
type ComplexType complex64
```

在本文档中代表一个复数类型。

func real

```
func real(c ComplexType) FloatType
```

返回复数c的实部。

func imag

```
func imag(c ComplexType) FloatType
```

返回复数c的虚部。

func complex

```
func complex(r, i FloatType) ComplexType
```

使用实部r和虚部i生成一个复数。

func new

```
func new(Type) *Type
```

内建函数new分配内存。其第一个实参为类型，而非值。其返回值为指向该类型的新分配的零值的指针。

func make

```
func make(Type, size IntegerType) Type
```

内建函数make分配并初始化一个类型为切片、映射、或通道的对象。其第一个实参为类型，而非值。make的返回类型与其参数相同，而非指向它的指针。其具体结果取决于具体的类型：

```
切片：size指定了其长度。该切片的容量等于其长度。切片支持第二个整数实参可用来指定
      它必须不小于其长度，因此 make([]int, 0, 10) 会分配一个长度为0，容量为1
映射：初始分配的创建取决于size，但产生的映射长度为0。size可以省略，这种情况下就
      小的起始大小。
通道：通道的缓存根据指定的缓存容量初始化。若 size为零或被省略，该信道即为无缓存
```

func cap

```
func cap(v Type) int
```

内建函数cap返回 v 的容量，这取决于具体类型：

```
数组：v中元素的数量，与 len(v) 相同
数组指针：*v中元素的数量，与len(v) 相同
切片：切片的容量（底层数组的长度）；若 v为nil，cap(v) 即为零
信道：按照元素的单元，相应信道缓存的容量；若v为nil，cap(v)即为零
```

func len

```
func len(v Type) int
```

内建函数len返回 v 的长度，这取决于具体类型：

```
数组：v中元素的数量
数组指针：*v中元素的数量（v为nil时panic）
切片、映射：v中元素的数量；若v为nil，len(v)即为零
字符串：v中字节的数量
通道：通道缓存中队列（未读取）元素的数量；若v为 nil，len(v)即为零
```

func append

```
func append(slice []Type, elems ...Type) []Type
```

内建函数append将元素追加到切片的末尾。若它有足够的容量，其目标就会重新切片以容纳新的元素。否则，就会分配一个新的基本数组。append返回更新后的切片，因此必须存储追加后的结果。

```
slice = append(slice, elem1, elem2)
slice = append(slice, anotherSlice...)
```

作为特例，可以向一个字节切片append字符串，如下：

```
slice = append([]byte("hello "), "world"...)
```

func copy

```
func copy(dst, src []Type) int
```

内建函数copy将元素从来源切片复制到目标切片中，也能将字节从字符串复制到字节切片中。copy返回被复制的元素数量，它会是 len(src) 和 len(dst) 中较小的那个。来源和目标的底层内存可以重叠。

func delete

```
func delete(m map[Type]Type1, key Type)
```

内建函数delete按照指定的键将元素从映射中删除。若m为nil或无此元素，delete不进行操作。

func close

```
func close(c chan<- Type)
```

内建函数close关闭信道，该通道必须为双向的或只发送的。它应当只由发送者执行，而不应由接收者执行，其效果是在最后发送的值被接收后停止该通道。在最后的值从已关闭的信道中被接收后，任何对其的接收操作都会无阻塞的成功。对于已关闭的信道，语句：

```
x, ok := <-c
```

还会将ok置为false。

func panic

```
func panic(v interface{})
```

内建函数panic停止当前Go程的正常执行。当函数F调用panic时，F的正常执行就会立刻停止。F中defer的所有函数先入后出执行后，F返回给其调用者G。G如同F一样行动，层层返回，直到该Go程中所有函数都按相反的顺序停止执行。之后，程序被终止，而错误情况会被报告，包括引发该恐慌的实参值，此终止序列称为恐慌过程。

func recover

```
func recover() interface{}
```

内建函数recover允许程序管理恐慌过程中的Go程。在defer的函数中，执行recover调用会取回传至panic调用的错误值，恢复正常执行，停止恐慌过程。若recover在defer的函数之外被调用，它将不会停止恐慌过程序列。在此情况下，或当该Go程不在恐慌过程中时，或提供给panic的实参为nil时，recover就会返回nil。

func print

```
func print(args ...Type)
```

内建函数print以特有的方法格式化参数并将结果写入标准错误，用于自举和调试。

func println

```
func println(args ...Type)
```

println类似print，但会在参数输出之间添加空格，输出结束后换行。

package bytes

```
import "bytes"
```

bytes包实现了操作[]byte的常用函数。本包的函数和strings包的函数相当类似。

Index

Examples

- Compare (Search)
- TrimPrefix
- TrimSuffix

Constants

```
const MinRead = 512
```

MinRead是被Buffer.ReadFrom传递给Read调用的最小尺寸。只要该Buffer在保存内容之外有最少MinRead字节的余量，其ReadFrom方法就不会增加底层的缓冲。

Variables

```
var ErrTooLarge = errors.New("bytes.Buffer: too large")
```

如果内存中不能申请足够保存数据的缓冲，ErrTooLarge就会被传递给panic函数。

func Compare

```
func Compare(a, b []byte) int
```

Compare函数返回一个整数表示两个[]byte切片按字典序比较的结果（类同C的strcmp）。如果a==b返回0；如果a<b返回-1；否则返回+1。nil参数视为空切片。

Example

```
// Interpret Compare's result by comparing it to zero.
var a, b []byte
if bytes.Compare(a, b) < 0 {
    // a less b
}
if bytes.Compare(a, b) <= 0 {
    // a less or equal b
}
if bytes.Compare(a, b) > 0 {
    // a greater b
}
if bytes.Compare(a, b) >= 0 {
    // a greater or equal b
}
// Prefer Equal to Compare for equality comparisons.
if bytes.Equal(a, b) {
    // a equal b
}
if !bytes.Equal(a, b) {
    // a not equal b
}
```

Example (Search)

```
// Binary search to find a matching byte slice.
var needle []byte
var haystack [][]byte // Assume sorted
i := sort.Search(len(haystack), func(i int) bool {
    // Return haystack[i] >= needle.
    return bytes.Compare(haystack[i], needle) >= 0
})
if i < len(haystack) && bytes.Equal(haystack[i], needle) {
    // Found it!
}
```

func Equal

```
func Equal(a, b []byte) bool
```

判断两个切片的内容是否完全相同。

func EqualFold

```
func EqualFold(s, t []byte) bool
```

判断两个utf-8编码切片（将unicode大写、小写、标题三种格式字符视为相同）是否相同。

func Runes

```
func Runes(s []byte) []rune
```

Runes函数返回和s等价的[]rune切片。（将utf-8编码的unicode码值分别写入单个rune）

func HasPrefix

```
func HasPrefix(s, prefix []byte) bool
```

判断s是否有前缀切片prefix。

func HasSuffix

```
func HasSuffix(s, suffix []byte) bool
```

判断s是否有后缀切片suffix。

func Contains

```
func Contains(b, subslice []byte) bool
```

判断切片b是否包含子切片subslice。

func Count

```
func Count(s, sep []byte) int
```

Count计算s中有多少个不重叠的sep子切片。

func Index

```
func Index(s, sep []byte) int
```

子切片sep在s中第一次出现的位置，不存在则返回-1。

func IndexByte

```
func IndexByte(s []byte, c byte) int
```

字符c在s中第一次出现的位置，不存在则返回-1。

func IndexRune

```
func IndexRune(s []byte, r rune) int
```

unicode字符r的utf-8编码在s中第一次出现的位置，不存在则返回-1。

func IndexAny

```
func IndexAny(s []byte, chars string) int
```

字符串chars中的任一utf-8编码在s中第一次出现的位置，如不存在或者chars为空字符串则返回-1

func IndexFunc

```
func IndexFunc(s []byte, f func(r rune) bool) int
```

s中第一个满足函数f的位置i（该处的utf-8码值r满足f(r)==true），不存在则返回-1

func LastIndex

```
func LastIndex(s, sep []byte) int
```

切片sep在字符串s中最后一次出现的位置，不存在则返回-1。

func LastIndexAny

```
func LastIndexAny(s []byte, chars string) int
```

字符串chars中的任一utf-8字符在s中最后一次出现的位置，如不存在或者chars为空字符串则返回-1。

func LastIndexFunc

```
func LastIndexFunc(s []byte, f func(r rune) bool) int
```

s中最后一个满足函数f的unicode码值的位置i，不存在则返回-1。

func Title

```
func Title(s []byte) []byte
```

返回s中每个单词的首字母都改为标题格式的拷贝。

BUG: Title用于划分单词的规则不能很好的处理Unicode标点符号。

func ToLower

```
func ToLower(s []byte) []byte
```

返回将所有字母都转为对应的小写版本的拷贝。

func ToLowerSpecial

```
func ToLowerSpecial(_case unicode.SpecialCase, s []byte) []byte
```

使用_case规定的字符映射，返回将所有字母都转为对应的小写版本的拷贝。

func ToUpper

```
func ToUpper(s []byte) []byte
```

返回将所有字母都转为对应的大写版本的拷贝。

func ToUpperSpecial

```
func ToUpperSpecial(_case unicode.SpecialCase, s []byte) []byte
```

使用_case规定的字符映射，返回将所有字母都转为对应的大写版本的拷贝。

func ToTitle

```
func ToTitle(s []byte) []byte
```

返回将所有字母都转为对应的标题版本的拷贝。

func ToTitleSpecial

```
func ToTitleSpecial(_case unicode.SpecialCase, s []byte) []byte
```

使用_case规定的字符映射，返回将所有字母都转为对应的标题版本的拷贝。

func Repeat

```
func Repeat(b []byte, count int) []byte
```

返回count个b串联形成的新的切片。

func Replace

```
func Replace(s, old, new []byte, n int) []byte
```

返回将s中前n个不重叠old切片序列都替换为new的新的切片拷贝，如果n<0会替换所有old子切片。

func Map

```
func Map(mapping func(r rune) rune, s []byte) []byte
```

将s的每一个unicode码值r都替换为mapping(r)，返回这些新码值组成的切片拷贝。如果mapping返回一个负值，将会丢弃该码值而不会被替换（返回值中对应位置将没有码值）。

func Trim

```
func Trim(s []byte, cutset string) []byte
```

返回将s前后端所有cutset包含的unicode码值都去掉的子切片。（共用底层数组）

func TrimSpace

```
func TrimSpace(s []byte) []byte
```

返回将s前后端所有空白（unicode.IsSpace指定）都去掉的子切片。（共用底层数组）

func TrimFunc

```
func TrimFunc(s []byte, f func(r rune) bool) []byte
```

返回将s前后端所有满足f的unicode码值都去掉的子切片。（共用底层数组）

func TrimLeft

```
func TrimLeft(s []byte, cutset string) []byte
```

返回将s前端所有cutset包含的unicode码值都去掉的子切片。（共用底层数组）

func TrimLeftFunc

```
func TrimLeftFunc(s []byte, f func(r rune) bool) []byte
```

返回将s前端所有满足f的unicode码值都去掉的子切片。（共用底层数组）

func TrimPrefix

```
func TrimPrefix(s, prefix []byte) []byte
```

返回去除s可能的前缀prefix的子切片。（共用底层数组）

Example

```
var b = []byte("Goodbye,, world!")
b = bytes.TrimPrefix(b, []byte("Goodbye,"))
b = bytes.TrimPrefix(b, []byte("See ya,"))
fmt.Printf("Hello%s", b)
```

Output:

```
Hello, world!
```

func TrimRight

```
func TrimRight(s []byte, cutset string) []byte
```

返回将s后端所有cutset包含的unicode码值都去掉的子切片。（共用底层数组）

func TrimRightFunc

```
func TrimRightFunc(s []byte, f func(r rune) bool) []byte
```

返回将s后端所有满足f的unicode码值都去掉的子切片。（共用底层数组）

func TrimSuffix

```
func TrimSuffix(s, suffix []byte) []byte
```

返回去除s可能的后缀suffix的子切片。（共用底层数组）

Example

```
var b = []byte("Hello, goodbye, etc!")
b = bytes.TrimSuffix(b, []byte("goodbye, etc!"))
b = bytes.TrimSuffix(b, []byte("gopher"))
b = append(b, bytes.TrimSuffix([]byte("world!"), []byte("x!"))...)
os.Stdout.Write(b)
```

Output:

```
Hello, world!
```

func Fields

```
func Fields(s []byte) [][]byte
```

返回将字符串按照空白（unicode.IsSpace确定，可以是一到多个连续的空白字符）分割的多个子切片。如果字符串全部是空白或者是空字符串的话，会返回空切片。

func FieldsFunc

```
func FieldsFunc(s []byte, f func(rune) bool) [][]byte
```

类似Fields，但使用函数f来确定分割符（满足f的utf-8码值）。如果字符串全部是分隔符或者是空字符串的话，会返回空切片。

func Split

```
func Split(s, sep []byte) [][]byte
```

用去掉s中出现的sep的方式进行分割，会分割到结尾，并返回生成的所有[]byte切片组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个[]byte切片。

func SplitN

```
func SplitN(s, sep []byte, n int) [][]byte
```

用去掉s中出现的sep的方式进行分割，会分割到最多n个子切片，并返回生成的所有[]byte切片组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个[]byte切片。参数n决定返回的切片的数目：

```
n > 0 : 返回的切片最多n个子字符串；最后一个子字符串包含未进行切割的部分。
n == 0: 返回nil
n < 0 : 返回所有的子字符串组成的切片
```

func SplitAfter

```
func SplitAfter(s, sep []byte) [][]byte
```

用从s中出现的sep后面切断的方式进行分割，会分割到结尾，并返回生成的所有[]byte切片组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个[]byte切片。

func SplitAfterN

```
func SplitAfterN(s, sep []byte, n int) [][]byte
```

用从s中出现的sep后面切断的方式进行分割，会分割到最多n个子切片，并返回生成的所有[]byte切片组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个[]byte切片。参数n决定返回的切片的数目：

```
n > 0 : 返回的切片最多n个子字符串；最后一个子字符串包含未进行切割的部分。
n == 0: 返回nil
n < 0 : 返回所有的子字符串组成的切片
```

func Join

```
func Join(s [][]byte, sep []byte) []byte
```

将一系列[]byte切片连接为一个[]byte切片，之间用sep来分隔，返回生成的新切片。

type Reader

```
type Reader struct {
    // 内含隐藏或非导出字段
}
```

Reader类型通过从一个[]byte读取数据，实现了io.Reader、io.Seeker、io.ReaderAt、io.WriterTo、io.ByteScanner、io.RuneScanner接口。

func NewReader

```
func NewReader(b []byte) *Reader
```

NewReader创建一个从s读取数据的Reader。

func (*Reader) Len

```
func (r *Reader) Len() int
```

Len返回r包含的切片中还没有被读取的部分。

func (*Reader) Read

```
func (r *Reader) Read(b []byte) (n int, err error)
```

func (*Reader) ReadByte

```
func (r *Reader) ReadByte() (b byte, err error)
```

func (*Reader) UnreadByte

```
func (r *Reader) UnreadByte() error
```

func (*Reader) ReadRune

```
func (r *Reader) ReadRune() (ch rune, size int, err error)
```

func (*Reader) UnreadRune

```
func (r *Reader) UnreadRune() error
```

func (*Reader) Seek

```
func (r *Reader) Seek(offset int64, whence int) (int64, error)
```

Seek实现了io.Seeker接口。

func (*Reader) ReadAt

```
func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)
```

func (*Reader) WriteTo

```
func (r *Reader) WriteTo(w io.Writer) (n int64, err error)
```

WriteTo实现了io.WriterTo接口。

type Buffer

```
type Buffer struct {
    // 内含隐藏或非导出字段
}
```

Buffer是一个实现了读写方法的可变大小的字节缓冲。本类型的零值是一个空的可用于读写的缓冲。

Example

```
var b bytes.Buffer // A Buffer needs no initialization.
b.Write([]byte("Hello "))
fmt.Fprintf(&b, "world!")
b.WriteTo(os.Stdout)
```

Output:

```
Hello world!
```

Example (Reader)

```
// A Buffer can turn a string or a []byte into an io.Reader.
buf := bytes.NewBufferString("R29waGVycyBydWxlIQ==")
dec := base64.NewDecoder(base64.StdEncoding, buf)
io.Copy(os.Stdout, dec)
```

Output:

```
Gophers rule!
```

func NewBuffer

```
func NewBuffer(buf []byte) *Buffer
```

NewBuffer使用buf作为初始内容创建并初始化一个Buffer。本函数用于创建一个用于读取已存在数据的buffer；也用于指定用于写入的内部缓冲的大小，此时，buf应为一个具有指定容量但长度为0的切片。buf会被作为返回值的底层缓冲切片。

大多数情况下，new(Buffer)（或只是声明一个Buffer类型变量）就足以初始化一个Buffer了。

func NewBufferString

```
func NewBufferString(s string) *Buffer
```

NewBuffer使用s作为初始内容创建并初始化一个Buffer。本函数用于创建一个用于读取已存在数据的buffer。

大多数情况下，new(Buffer)（或只是声明一个Buffer类型变量）就足以初始化一个Buffer了。

func (*Buffer) Reset

```
func (b *Buffer) Reset()
```

Reset重设缓冲，因此会丢弃全部内容，等价于b.Truncate(0)。

func (*Buffer) Len

```
func (b *Buffer) Len() int
```

返回缓冲中未读取部分的字节长度；b.Len() == len(b.Bytes())。

func (*Buffer) Bytes

```
func (b *Buffer) Bytes() []byte
```

返回未读取部分字节数据的切片，len(b.Bytes()) == b.Len()。如果中间没有调用其他方法，修改返回的切片的内容会直接改变Buffer的内容。

func (*Buffer) String

```
func (b *Buffer) String() string
```

将未读取部分的字节数据作为字符串返回，如果b是nil指针，会返回"<nil>"。

func (*Buffer) Truncate

```
func (b *Buffer) Truncate(n int)
```

丢弃缓冲中除前n字节数据外的其它数据，如果n小于零或者大于缓冲容量将panic。

func (*Buffer) Grow

```
func (b *Buffer) Grow(n int)
```

必要时会增加缓冲的容量，以保证n字节的剩余空间。调用Grow(n)后至少可以向缓冲中写入n字节数据而无需申请内存。如果n小于零或者不能增加容量都会panic。

func (*Buffer) Read

```
func (b *Buffer) Read(p []byte) (n int, err error)
```

Read方法从缓冲中读取数据直到缓冲中没有数据或者读取了len(p)字节数据，将读取的数据写入p。返回值n是读取的字节数，除非缓冲中完全没有数据可以读取并写入p，此时返回值err为io.EOF；否则err总是nil。

func (*Buffer) Next

```
func (b *Buffer) Next(n int) []byte
```

返回未读取部分前n字节数据的切片，并且移动读取位置，就像调用了Read方法一样。如果缓冲内数据不足，会返回整个数据的切片。切片只在下一次调用b的读/写方法前才合法。

func (*Buffer) ReadByte

```
func (b *Buffer) ReadByte() (c byte, err error)
```

ReadByte读取并返回缓冲中的下一个字节。如果没有数据可用，返回值err为io.EOF。

func (*Buffer) UnreadByte

```
func (b *Buffer) UnreadByte() error
```

UnreadByte吐出最近一次读取操作读取的最后一个字节。如果最后一次读取操作之后进行了写入，本方法会返回错误。

func (*Buffer) ReadRune

```
func (b *Buffer) ReadRune() (r rune, size int, err error)
```

ReadRune读取并返回缓冲中的下一个utf-8码值。如果没有数据可用，返回值err为io.EOF。如果缓冲中的数据是错误的utf-8编码，本方法会吃掉一字节并返回(U+FFFD, 1, nil)。

func (*Buffer) UnreadRune

```
func (b *Buffer) UnreadRune() error
```

UnreadRune吐出最近一次调用ReadRune方法读取的unicode码值。如果最近一次读写操作不是ReadRune，本方法会返回错误。（这里就能看出来UnreadRune比UnreadByte严格多了）

func (*Buffer) ReadBytes

```
func (b *Buffer) ReadBytes(delim byte) (line []byte, err error)
```

ReadBytes读取直到第一次遇到delim字节，返回一个包含已读取的数据和delim字节的切片。如果ReadBytes方法在读取到delim之前遇到了错误，它会返回在错误之前读取的数据以及该错误（一般是io.EOF）。当且仅当ReadBytes方法返回的切片不以delim结尾时，会返回一个非nil的错误。

func (*Buffer) ReadString

```
func (b *Buffer) ReadString(delim byte) (line string, err error)
```

ReadString读取直到第一次遇到delim字节，返回一个包含已读取的数据和delim字节的字符串。如果ReadString方法在读取到delim之前遇到了错误，它会返回在错误之前读取的数据以及该错误（一般是io.EOF）。当且仅当ReadString方法返回的切片不以delim结尾时，会返回一个非nil的错误。

func (*Buffer) Write

```
func (b *Buffer) Write(p []byte) (n int, err error)
```

Write将p的内容写入缓冲中，如必要会增加缓冲容量。返回值n为len(p)，err总是nil。如果缓冲变得太大，Write会采用错误值ErrTooLarge引发panic。

func (*Buffer) WriteString

```
func (b *Buffer) WriteString(s string) (n int, err error)
```

Write将s的内容写入缓冲中，如必要会增加缓冲容量。返回值n为len(p)，err总是nil。如果缓冲变得太大，Write会采用错误值ErrTooLarge引发panic。

func (*Buffer) WriteByte

```
func (b *Buffer) WriteByte(c byte) error
```

WriteByte将字节c写入缓冲中，如必要会增加缓冲容量。返回值总是nil，但仍保留以匹配bufio.Writer的WriteByte方法。如果缓冲太大，WriteByte会采用错误值ErrTooLarge引发panic。

func (*Buffer) WriteRune

```
func (b *Buffer) WriteRune(r rune) (n int, err error)
```

WriteByte将unicode码值r的utf-8编码写入缓冲中，如必要会增加缓冲容量。返回值总是nil，但仍保留以匹配bufio.Writer的WriteRune方法。如果缓冲太大，WriteRune会采用错误值ErrTooLarge引发panic。

func (*Buffer) ReadFrom

```
func (b *Buffer) ReadFrom(r io.Reader) (n int64, err error)
```

ReadFrom从r中读取数据直到结束并将读取的数据写入缓冲中，如必要会增加缓冲容量。返回值n为从r读取并写入b的字节数；会返回读取时遇到的除了io.EOF之外的错误。如果缓冲太大，ReadFrom会采用错误值ErrTooLarge引发panic。

func (*Buffer) WriteTo

```
func (b *Buffer) WriteTo(w io.Writer) (n int64, err error)
```

WriteTo从缓冲中读取数据直到缓冲内没有数据或遇到错误，并将这些数据写入w。返回值n为从b读取并写入w的字节数；返回值总是可以无溢出的写入int类型，但为了匹配io.WriterTo接口设为int64类型。从b读取是遇到的非io.EOF错误及写入w时遇到的错误都会终止本方法并返回该错误。

package compress

package bzip2

```
import "compress/bzip2"
```

bzip2包实现bzip2的解压缩。

Index

type StructuralError

```
type StructuralError string
```

当bzip2数据的语法不合法时，会返回本类型错误。

func (StructuralError) Error

```
func (s StructuralError) Error() string
```

func NewReader

```
func NewReader(r io.Reader) io.Reader
```

NewReader返回一个从r读取bzip2压缩数据并解压缩后返回给调用者的io.Reader。

package flate

`import "compress/flate"`

flate包实现了deflate压缩数据格式，参见RFC 1951。gzip包和zlib包实现了对基于deflate的文件格式的访问。

Index

Constants

```
const (
    NoCompression = 0
    BestSpeed     = 1
    BestCompression    = 9
    DefaultCompression = -1
)
```

type CorruptInputError

```
type CorruptInputError int64
```

CorruptInputError表示在输入的指定偏移量位置存在损坏。

func (CorruptInputError) Error

```
func (e CorruptInputError) Error() string
```

type InternalError

```
type InternalError string
```

InternalError表示flate数据自身的错误。

func (InternalError) Error

```
func (e InternalError) Error() string
```

type ReadError

```
type ReadError struct {
    Offset int64 // 错误出现的位置（字节偏移量）
    Err    error // 下层的读取操作返回的错误
}
```

ReadError代表在读取输入流时遇到的错误。

func (*ReadError) Error

```
func (e *ReadError) Error() string
```

type WriteError

```
type WriteError struct {
    Offset int64 // 错误出现的位置（字节偏移量）
    Err    error // 下层的写入操作返回的错误
}
```

WriteError代表在写入输出流时遇到的错误。

func (*WriteError) Error

```
func (e *WriteError) Error() string
```

type Reader

```
type Reader interface {
    io.Reader
    io.ByteReader
}
```

NewReader真正需要的接口。如果提供的Io.Reader没有提供ReadByte方法，NewReader函数会自行添加缓冲。

func NewReader

```
func NewReader(r io.Reader) io.ReadCloser
```

NewReader返回一个从r读取并解压数据的io.ReadCloser。调用者有责任在读取完毕后调用返回值的Close方法。

func NewReaderDict

```
func NewReaderDict(r io.Reader, dict []byte) io.ReadCloser
```

NewReaderDict类似NewReader，但会使用预设的字典初始化返回的Reader。

返回的Reader表现的好像原始未压缩的数据流以该字典起始（并已经被读取）。NewReaderDict用于读取NewWriterDict压缩的数据。

type Writer

```
type Writer struct {
    // 内含隐藏或非导出字段
}
```

Writer将提供给它的数据压缩后写入下层的io.Writer接口。

func NewWriter

```
func NewWriter(w io.Writer, level int) (*Writer, error)
```

NewWriter返回一个压缩水平为level的Writer。

和zlib包一样，level的范围是1（BestSpeed）到9 （BestCompression）。值越大，压缩效果越好，但也越慢；level为0表示不尝试做任何压缩，只添加必需的deflate框架；level为-1时会使用默认的压缩水平；如果level在[-1, 9]范围内，error返回值将是nil，否则将返回非nil的错误值。

func NewWriterDict

```
func NewWriterDict(w io.Writer, level int, dict []byte) (*Writer, e
```

NewWriterDict类似NewWriter，但会使用预设的字典初始化返回的Writer。

返回的Writer表现的好像已经将原始、未压缩数据dict（压缩后未产生任何数据的）写入w了，使用w压缩的数据只能被使用同样的字典初始化生成的Reader接口解压缩。（类似加解密的初始向量/密钥）

func (*Writer) Reset

```
func (w *Writer) Reset(dst io.Writer)
```

Reset将w重置，丢弃当前的写入状态，并将下层输出目标设为dst。效果上等价于将w设为使用dst和w的压缩水平、字典重新调用NewWriter或NewWriterDict返回的*Writer。

func (*Writer) Write

```
func (w *Writer) Write(data []byte) (n int, err error)
```

Write向w写入数据，最终会将压缩后的数据写入下层io.Writer接口。

func (*Writer) Flush

```
func (w *Writer) Flush() error
```

Flush将缓冲中的压缩数据刷新到下层io.Writer接口中。

本方法主要用在传输压缩数据的网络连接中，以保证远端的接收者可以获得足够的数据来重构数据报。Flush会阻塞直到所有缓冲中的数据都写入下层io.Writer接口后才返回。如果下层的io.Writetr接口返回一个错误，Flush也会返回该错误。在zlib包的术语中，Flush方法等价于Z_SYNC_FLUSH。

func (*Writer) Close

```
func (w *Writer) Close() error
```

Close刷新缓冲并关闭w。

package gzip

`import "compress/gzip"`

gzip包实现了gzip格式压缩文件的读写，参见RFC 1952。

Index

Constants

```
const (
    NoCompression      = flate.NoCompression
    BestSpeed          = flate.BestSpeed
    BestCompression    = flate.BestCompression
    DefaultCompression = flate.DefaultCompression
)
```

这些常量都是拷贝自flate包，因此导入"compress/gzip"后，就不必再导入"compress/flate"了。

Variables

```
var (
    // 当读取的gzip数据的校验和错误时，会返回ErrChecksum
    ErrChecksum = errors.New("gzip: invalid checksum")
    // 当读取的gzip数据的头域错误时，会返回ErrHeader
    ErrHeader = errors.New("gzip: invalid header")
)
```

type Header

```
type Header struct {
    Comment string    // 注释
    Extra   []byte    // 额外数据
    ModTime time.Time // 修改时间
    Name    string    // 文件名
    OS      byte      // 操作系统类型
}
```

gzip文件保存一个头域，提供关于被压缩的文件的一些元数据。该头域作为Writer和Reader类型的一个可导出字段，可以提供给调用者访问。

type Reader

```
type Reader struct {
    Header
    // 内含隐藏或非导出字段
}
```

Reader类型满足io.Reader接口，可以从gzip格式压缩文件读取并解压数据。

一般，一个gzip文件可以是多个gzip文件的串联，每一个都有自己的头域。从Reader读取数据会返回串联的每个文件的解压数据，但只有第一个文件的头域被记录在Reader的Header字段里。

gzip文件会保存未压缩数据的长度与校验和。当读取到未压缩数据的结尾时，如果数据的长度或者校验和不正确，Reader会返回ErrCheckSum。因此，调用者应该将Read方法返回的数据视为暂定的，直到他们在数据结尾获得了一个io.EOF。

func NewReader

```
func NewReader(r io.Reader) (*Reader, error)
```

NewReader返回一个从r读取并解压数据的*Reader。其实现会缓冲输入流的数据，并可能从r中读取比需要的更多的数据。调用者有责任在读取完毕后调用返回值的Close方法。

func (*Reader) Reset

```
func (z *Reader) Reset(r io.Reader) error
```

Reset将z重置，丢弃当前的读取状态，并将下层读取目标设为r。效果上等价于将z设为使用r重新调用NewReader返回的Reader。这让我们可以重用z而不是再申请一个新的。（因此效率更高）

func (*Reader) Read

```
func (z *Reader) Read(p []byte) (n int, err error)
```

func (*Reader) Close

```
func (z *Reader) Close() error
```

调用Close会关闭z，但不会关闭下层io.Reader接口。

type Writer

```
type Writer struct {
    Header
    // 内含隐藏或非导出字段
}
```

Writer满足io.WriteCloser接口。它会将提供给它的数据压缩后写入下层io.Writer接口。

func NewWriter

```
func NewWriter(w io.Writer) *Writer
```

NewWriter创建并返回一个Writer。写入返回值的数据都会在压缩后写入w。调用者有责任在结束写入后调用返回值的Close方法。因为写入的数据可能保存在缓冲中没有刷新入下层。

如要设定Writer.Header字段，调用者必须在第一次调用Write方法或者Close方法之前设置。Header字段的Comment和Name字段是go的utf-8字符串，但下层格式要求为NUL中止的ISO 8859-1 (Latin-1)序列。如果这两个字段的字符串包含NUL或非Latin-1字符，将导致Write方法返回错误。

func NewWriterLevel

```
func NewWriterLevel(w io.Writer, level int) (*Writer, error)
```

NewWriterLevel类似NewWriter但指定了压缩水平而不是采用默认的DefaultCompression。

参数level可以是DefaultCompression、NoCompression或BestSpeed与BestCompression之间包括二者的任何整数。如果level合法，返回的错误值为nil。

func (*Writer) Reset

```
func (z *Writer) Reset(w io.Writer)
```

Reset将z重置，丢弃当前的写入状态，并将下层输出目标设为dst。效果上等价于将w设为使用dst和w的压缩水平重新调用NewWriterLevel返回的*Writer。这让我们可以重用z而不是再申请一个新的。（因此效率更高）

func (*Writer) Write

```
func (z *Writer) Write(p []byte) (int, error)
```

Write将p压缩后写入下层io.Writer接口。压缩后的数据不一定会立刻刷新，除非Writer被关闭或者显式的刷新。

func (*Writer) Flush

```
func (z *Writer) Flush() error
```

Flush将缓冲中的压缩数据刷新到下层io.Writer接口中。

本方法主要用在传输压缩数据的网络连接中，以保证远端的接收者可以获得足够的数据来重构数据报。Flush会阻塞直到所有缓冲中的数据都写入下层io.Writer接口后才返回。如果下层的io.Writetr接口返回一个错误，Flush也会返回该错误。在zlib包的术语中，Flush方法等价于Z_SYNC_FLUSH。

func (*Writer) Close

```
func (z *Writer) Close() error
```

调用Close会关闭z，但不会关闭下层io.Writer接口。

package lzw

```
import "compress/lzw"
```

lzw包实现了Lempel-Ziv-Welch数据压缩格式，这是一种T. A. Welch在“A Technique for High-Performance Data Compression”一文（Computer, 17(6) (June 1984), pp 8-19）提出的一种压缩格式。

本包实现了用于GIF、TIFF、PDF文件的lzw压缩格式，这是一种最长达到12位的变长码，头两个非字面码为clear和EOF码。

Index

type Order

```
type Order int
```

Order指定一个lzw数据流的位顺序。

```
const (
    // LSB表示最小权重位在前，用在GIF文件格式
    LSB Order = iota
    // MSB表示最大权重位在前，用在TIFF和PDF文件格式
    MSB
)
```

func NewReader

```
func NewReader(r io.Reader, order Order, litWidth int) io.ReadClose
```

创建一个io.ReadCloser，它从r读取并解压数据。调用者有责任在结束读取后调用返回值的Close方法；litWidth指定字面码的位数，必须在[2,8]范围内，一般为8。

func NewWriter

```
func NewWriter(w io.Writer, order Order, litWidth int) io.WriteClos
```

创建一个io.WriteCloser，它将数据压缩后写入w。调用者有责任在结束写入后调用返回值的Close方法；litWidth指定字面码的位数，必须在[2,8]范围内，一般为8。

package zlib

```
import "compress/zlib"
```

zlib包实现了对zlib格式压缩数据的读写，参见RFC 1950。

本包的实现提供了在读取时解压和写入时压缩的滤镜。例如，将压缩数据写入一个bytes.Buffer：

```
var b bytes.Buffer
w := zlib.NewWriter(&b)
w.Write([]byte("hello, world\n"))
w.Close()
```

然后将数据读取回来：

```
r, err := zlib.NewReader(&b)
io.Copy(os.Stdout, r)
r.Close()
```

Index

Examples

Constants

```
const (
    NoCompression      = flate.NoCompression
    BestSpeed          = flate.BestSpeed
    BestCompression    = flate.BestCompression
    DefaultCompression = flate.DefaultCompression
)
```

这些常量都是拷贝自flate包，因此导入"compress/zlib"后，就不必再导入"compress/flate"了。

Variables

```
var (
    // 当读取的zlib数据的校验和错误时，会返回ErrChecksum
    ErrChecksum = errors.New("zlib: invalid checksum")
    // 当读取的zlib数据的目录错误时，会返回ErrDictionary
    ErrDictionary = errors.New("zlib: invalid dictionary")
    // 当读取的zlib数据的头域错误时，会返回ErrHeader
    ErrHeader = errors.New("zlib: invalid header")
)
```

func NewReader

```
func NewReader(r io.Reader) (io.ReadCloser, error)
```

NewReader返回一个从r读取并解压数据的io.ReadCloser。其实现会缓冲输入流的数据，并可能从r中读取比需要的更多的数据。调用者有责任在读取完毕后调用返回值的Close方法。

Example

```
buff := []byte{120, 156, 202, 72, 205, 201, 201, 215, 81, 40, 207,
    47, 202, 73, 225, 2, 4, 0, 0, 255, 255, 33, 231, 4, 147}
b := bytes.NewReader(buff)
r, err := zlib.NewReader(b)
if err != nil {
    panic(err)
}
io.Copy(os.Stdout, r)
```

Output:

```
hello, world
```

func NewReaderDict

```
func NewReaderDict(r io.Reader, dict []byte) (io.ReadCloser, error)
```

NewReaderDict类似NewReader，但会使用预设的字典初始化返回的Reader。

如果压缩数据没有采用字典，本函数会忽略该参数。

type Writer

```
type Writer struct {
    // 内含隐藏或非导出字段
}
```

Writer将提供给它的数据压缩后写入下层io.Writer接口。

func NewWriter

```
func NewWriter(w io.Writer) *Writer
```

NewWriter创建并返回一个Writer。写入返回值的数据都会在压缩后写入w。

调用者有责任在结束写入后调用返回值的Close方法。因为写入的数据可能保存在缓冲中没有刷新入下层。

Example

```
var b bytes.Buffer
w := zlib.NewWriter(&b)
w.Write([]byte("hello, world\n"))
w.Close()
fmt.Println(b.Bytes())
```

Output:

```
[120 156 202 72 205 201 201 215 81 40 207 47 202 73 225 2 4 0 0 255
```

func NewWriterLevel

```
func NewWriterLevel(w io.Writer, level int) (*Writer, error)
```

NewWriterLevel类似NewWriter但指定了压缩水平而不是采用默认的DefaultCompression。

参数level可以是DefaultCompression、NoCompression或BestSpeed与BestCompression之间包括二者的任何整数。如果level合法，返回的错误值为nil。

func NewWriterLevelDict

```
func NewWriterLevelDict(w io.Writer, level int, dict []byte) (*Writ
```

NewWriterLevelDict类似NewWriterLevel但还指定了用于压缩的字典。dict参数可以为nil；否则，在返回的Writer关闭之前，其内容不可被修改。

func (*Writer) Reset

```
func (z *Writer) Reset(w io.Writer)
```

Reset将w重置，丢弃当前的写入状态，并将下层输出目标设为dst。效果上等价于将w设为使用dst和w的压缩水平、字典重新调用NewWriterLevel或NewWriterLevelDict返回的*Writer。

func (*Writer) Write

```
func (z *Writer) Write(p []byte) (n int, err error)
```

Write将p压缩后写入下层io.Writer接口。压缩后的数据不一定会立刻刷新，除非Writer被关闭或者显式的刷新。

func (*Writer) Flush

```
func (z *Writer) Flush() error
```

Flush将缓冲中的压缩数据刷新到下层io.Writer接口中。

func (*Writer) Close

```
func (z *Writer) Close() error
```

调用Close会刷新缓冲并关闭w，但不会关闭下层io.Writer接口。

package container

package heap

```
import "container/heap"
```

heap包提供了对任意类型（实现了heap.Interface接口）的堆操作。（最小）堆是具有“每个节点都是以其为根的子树中最小值”属性的树。

树的最小元素为其根元素，索引0的位置。

heap是常用的实现优先队列的方法。要创建一个优先队列，实现一个具有使用（负的）优先级作为比较的依据的Less方法的Heap接口，如此一来可用Push添加项目而用Pop取出队列最高优先级的项目。

Example (IntHeap)

```
// This example demonstrates an integer heap built using the heap i
package heap_test
import (
    "container/heap"
    "fmt"
)
// An IntHeap is a min-heap of ints.
type IntHeap []int
func (h IntHeap) Len() int           { return len(h) }
func (h IntHeap) Less(i, j int) bool { return h[i] < h[j] }
func (h IntHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }
func (h *IntHeap) Push(x interface{}) {
    // Push and Pop use pointer receivers because they modify the s
    // not just its contents.
    *h = append(*h, x.(int))
}
func (h *IntHeap) Pop() interface{} {
    old := *h
    n := len(old)
    x := old[n-1]
    *h = old[0 : n-1]
    return x
}
// This example inserts several ints into an IntHeap, checks the mi
// and removes them in order of priority.
func Example_intHeap() {
    h := &IntHeap{2, 1, 5}
    heap.Init(h)
    heap.Push(h, 3)
    fmt.Printf("minimum: %d\n", (*h)[0])
    for h.Len() > 0 {
        fmt.Printf("%d ", heap.Pop(h))
    }
    // Output:
    // minimum: 1
    // 1 2 3 5
}
```

Example (PriorityQueue)

```
// This example demonstrates a priority queue built using the heap
package heap_test
import (
    "container/heap"
    "fmt"
)
// An Item is something we manage in a priority queue.
type Item struct {
    value    string // The value of the item; arbitrary.
    priority int    // The priority of the item in the queue.
```

```
    // The index is needed by update and is maintained by the heap.
    index int // The index of the item in the heap.
}
// A PriorityQueue implements heap.Interface and holds Items.
type PriorityQueue []*Item
func (pq PriorityQueue) Len() int { return len(pq) }
func (pq PriorityQueue) Less(i, j int) bool {
    // We want Pop to give us the highest, not lowest, priority so
    return pq[i].priority > pq[j].priority
}
func (pq PriorityQueue) Swap(i, j int) {
    pq[i], pq[j] = pq[j], pq[i]
    pq[i].index = i
    pq[j].index = j
}
func (pq *PriorityQueue) Push(x interface{}) {
    n := len(*pq)
    item := x.(*Item)
    item.index = n
    *pq = append(*pq, item)
}
func (pq *PriorityQueue) Pop() interface{} {
    old := *pq
    n := len(old)
    item := old[n-1]
    item.index = -1 // for safety
    *pq = old[0 : n-1]
    return item
}
// update modifies the priority and value of an Item in the queue.
func (pq *PriorityQueue) update(item *Item, value string, priority
    item.value = value
    item.priority = priority
    heap.Fix(pq, item.index)
}
// This example creates a PriorityQueue with some items, adds and r
// and then removes the items in priority order.
func Example_priorityQueue() {
    // Some items and their priorities.
    items := map[string]int{
        "banana": 3, "apple": 2, "pear": 4,
    }
    // Create a priority queue, put the items in it, and
    // establish the priority queue (heap) invariants.
    pq := make(PriorityQueue, len(items))
    i := 0
    for value, priority := range items {
        pq[i] = &Item{
            value:    value,
            priority: priority,
            index:    i,
        }
        i++
```

```
    }
    heap.Init(&pq)
    // Insert a new item and then modify its priority.
    item := &Item{
        value:    "orange",
        priority: 1,
    }
    heap.Push(&pq, item)
    pq.update(item, item.value, 5)
    // Take the items out; they arrive in decreasing priority orde
    for pq.Len() > 0 {
        item := heap.Pop(&pq).(*Item)
        fmt.Printf("%.2d:%s ", item.priority, item.value)
    }
    // Output:
    // 05:orange 04:pear 03:banana 02:apple
}
```

Index

Examples

type Interface

```
type Interface interface {
    sort.Interface
    Push(x interface{}) // 向末尾添加元素
    Pop() interface{}   // 从末尾删除元素
}
```

任何实现了本接口的类型都可以用于构建最小堆。最小堆可以通过heap.Init建立，数据是递增顺序或者空的话也是最小堆。最小堆的约束条件是：

```
!h.Less(j, i) for 0 <= i < h.Len() and 2*i+1 <= j <= 2*i+2 and j <
```

注意接口的Push和Pop方法是供heap包调用的，请使用heap.Push和heap.Pop来向一个堆添加或者删除元素。

func Init

```
func Init(h Interface)
```

一个堆在使用任何堆操作之前应先初始化。Init函数对于堆的约束性是幂等的（多次执行无意义），并可能在任何时候堆的约束性被破坏时被调用。本函数复杂度为O(n)，其中n等于h.Len()。

func Push

```
func Push(h Interface, x interface{})
```

向堆h中插入元素x，并保持堆的约束性。复杂度O(log(n))，其中n等于h.Len()。

func Pop

```
func Pop(h Interface) interface{}
```

删除并返回堆h中的最小元素（不影响约束性）。复杂度O(log(n))，其中n等于h.Len()。等价于Remove(h, 0)。

func Remove

```
func Remove(h Interface, i int) interface{}
```

删除堆中的第i个元素，并保持堆的约束性。复杂度O(log(n))，其中n等于h.Len()。

func Fix

```
func Fix(h Interface, i int)
```

在修改第i个元素后，调用本函数修复堆，比删除第i个元素后插入新元素更有效率。

复杂度O(log(n))，其中n等于h.Len()。

package list

```
import "container/list"
```

list包实现了双向链表。要遍历一个链表：

```
for e := l.Front(); e != nil; e = e.Next() {
    // do something with e.Value
}
```

Example

```
// Create a new list and put some numbers in it.
l := list.New()
e4 := l.PushBack(4)
e1 := l.PushFront(1)
l.InsertBefore(3, e4)
l.InsertAfter(2, e1)
// Iterate through list and print its contents.
for e := l.Front(); e != nil; e = e.Next() {
    fmt.Println(e.Value)
}
```

Output:

```
1
2
3
4
```

Index

Examples

type Element

```
type Element struct {
    // 元素保管的值
    Value interface{}
    // 内含隐藏或非导出字段
}
```

Element类型代表是双向链表的一个元素。

func (*Element) Next

```
func (e *Element) Next() *Element
```

Next返回链表的后一个元素或者nil。

func (*Element) Prev

```
func (e *Element) Prev() *Element
```

Prev返回链表的前一个元素或者nil。

type List

```
type List struct {
    // 内含隐藏或非导出字段
}
```

List代表一个双向链表。List零值为一个空的、可用的链表。

func New

```
func New() *List
```

New创建一个链表。

func (*List) Init

```
func (l *List) Init() *List
```

Init清空链表。

func (*List) Len

```
func (l *List) Len() int
```

Len返回链表中元素的个数，复杂度O(1)。

func (*List) Front

```
func (l *List) Front() *Element
```

Front返回链表第一个元素或nil。

func (*List) Back

```
func (l *List) Back() *Element
```

Back返回链表最后一个元素或nil。

func (*List) PushFront

```
func (l *List) PushFront(v interface{}) *Element
```

PushBack将一个值为v的新元素插入链表的第一个位置，返回生成的新元素。

func (*List) PushFrontList

```
func (l *List) PushFrontList(other *List)
```

PushFrontList创建链表other的拷贝，并将拷贝的最后一个位置连接到链表l的第一个位置。

func (*List) PushBack

```
func (l *List) PushBack(v interface{}) *Element
```

PushBack将一个值为v的新元素插入链表的最后一个位置，返回生成的新元素。

func (*List) PushBackList

```
func (l *List) PushBackList(other *List)
```

PushBack创建链表other的拷贝，并将链表l的最后一个位置连接到拷贝的第一个位置。

func (*List) InsertBefore

```
func (l *List) InsertBefore(v interface{}, mark *Element) *Element
```

InsertDefore将一个值为v的新元素插入到mark前面，并返回生成的新元素。如果mark不是l的元素，l不会被修改。

func (*List) InsertAfter

```
func (l *List) InsertAfter(v interface{}, mark *Element) *Element
```

InsertAfter将一个值为v的新元素插入到mark后面，并返回新生成的元素。如果mark不是l的元素，l不会被修改。

func (*List) MoveToFront

```
func (l *List) MoveToFront(e *Element)
```

MoveToFront将元素e移动到链表的第一个位置，如果e不是l的元素，l不会被修改。

func (*List) MoveToBack

```
func (l *List) MoveToBack(e *Element)
```

MoveToBack将元素e移动到链表的最后一个位置，如果e不是l的元素，l不会被修改。

func (*List) MoveBefore

```
func (l *List) MoveBefore(e, mark *Element)
```

MoveBefore将元素e移动到mark的前面。如果e或mark不是l的元素，或者e==mark，l不会被修改。

func (*List) MoveAfter

```
func (l *List) MoveAfter(e, mark *Element)
```

MoveAfter将元素e移动到mark的后面。如果e或mark不是l的元素，或者e==mark，l不会被修改。

func (*List) Remove

```
func (l *List) Remove(e *Element) interface{}
```

Remove删除链表中的元素e，并返回e.Value。

package ring

```
import "container/ring"
```

ring实现了环形链表的操作。

Index

type Ring

```
type Ring struct {
    Value interface{} // 供调用者使用，本包不会操作该字段
    // 包含隐藏或非导出字段
}
```

Ring类型代表环形链表的一个元素，同时也代表链表本身。环形链表没有头尾；指向环形链表任一元素的指针都可以作为整个环形链表看待。Ring零值是具有一个（Value字段为nil的）元素的链表。

func New

```
func New(n int) *Ring
```

New创建一个具有n个元素的环形链表。

func (*Ring) Len

```
func (r *Ring) Len() int
```

Len返回环形链表中的元素个数，复杂度O(n)。

func (*Ring) Next

```
func (r *Ring) Next() *Ring
```

返回后一个元素，r不能为空。

func (*Ring) Prev

```
func (r *Ring) Prev() *Ring
```

返回前一个元素，r不能为空。

func (*Ring) Move

```
func (r *Ring) Move(n int) *Ring
```

返回移动n个位置（n>=0向前移动，n<0向后移动）后的元素，r不能为空。

func (*Ring) Link

```
func (r *Ring) Link(s *Ring) *Ring
```

Link连接r和s，并返回r原本的后继元素r.Next()。r不能为空。

如果r和s指向同一个环形链表，则会删除掉r和s之间的元素，删掉的元素构成一个子链表，返回指向该子链表的指针（r的原后继元素）；如果没有删除元素，则仍然返回r的原后继元素，而不是nil。如果r和s指向不同的链表，将创建一个单独的链表，将s指向的链表插入r后面，返回s原最后一个元素后面的元素（即r的原后继元素）。

func (*Ring) Unlink

```
func (r *Ring) Unlink(n int) *Ring
```

删除链表中n % r.Len()个元素，从r.Next()开始删除。如果n % r.Len() == 0，不修改r。返回删除的元素构成的链表，r不能为空。

func (*Ring) Do

```
func (r *Ring) Do(f func(interface{}))
```

对链表的每一个元素都执行f（正向顺序），注意如果f改变了*r，Do的行为是未定义的。

package crypto

`import "crypto"`

crypto包搜集了常用的密码（算法）常量。

Index

type PublicKey

```
type PublicKey interface{}
```

代表一个使用未指定算法的公钥。

type PrivateKey

```
type PrivateKey interface{}
```

代表一个使用未指定算法的私钥。

type Hash

```
type Hash uint
```

Hash用来识别/标识另一个包里实现的加密函数。

```
const (
    MD4       Hash = 1 + iota // 导入code.google.com/p/go.crypto/md
    MD5                       // 导入crypto/md5
    SHA1                      // 导入crypto/sha1
    SHA224                    // 导入crypto/sha256
    SHA256                    // 导入crypto/sha256
    SHA384                    // 导入crypto/sha512
    SHA512                    // 导入crypto/sha512
    MD5SHA1                   // 未实现；MD5+SHA1用于TLS RSA
    RIPEMD160                 // 导入code.google.com/p/go.crypto/ri
)
```

func (Hash) Available

```
func (h Hash) Available() bool
```

报告是否有hash函数注册到该标识值。

func (Hash) Size

```
func (h Hash) Size() int
```

返回给定hash函数返回值的字节长度。

func (Hash) New

```
func (h Hash) New() hash.Hash
```

创建一个使用给定hash函数的hash.Hash接口，如果该标识值未注册hash函数，将会panic。

func RegisterHash

```
func RegisterHash(h Hash, f func() hash.Hash)
```

注册一个返回给定hash接口实例的函数，并指定其标识值，该函数应在实现hash接口的包的init函数中调用。

package aes

```
import "crypto/aes"
```

aes包实现了AES加密算法，参见U.S. Federal Information Processing Standards Publication 197。

Index

Constants

```
const BlockSize = 16
```

AES字节块大小。

type KeySizeError

```
type KeySizeError int
```

func (KeySizeError) Error

```
func (k KeySizeError) Error() string
```

func NewCipher

```
func NewCipher(key []byte) (cipher.Block, error)
```

创建一个cipher.Block接口。参数key为密钥，长度只能是16、24、32字节，用以选择AES-128、AES-192、AES-256。

package cipher

```
import "crypto/cipher"
```

cipher包实现了多个标准的用于包装底层块加密算法的加密算法实现。

参见http://csrc.nist.gov/groups/ST/toolkit/BCM/current_modes.html和NIST Special Publication 800-38A。

Index

Examples

type Block

```
type Block interface {
    // 返回加密字节块的大小
    BlockSize() int
    // 加密src的第一块数据并写入dst，src和dst可指向同一内存地址
    Encrypt(dst, src []byte)
    // 解密src的第一块数据并写入dst，src和dst可指向同一内存地址
    Decrypt(dst, src []byte)
}
```

Block接口代表一个使用特定密钥的底层块加/解密器。它提供了加密和解密独立数据块的能力。

type BlockMode

```
type BlockMode interface {
    // 返回加密字节块的大小
    BlockSize() int
    // 加密或解密连续的数据块，src的尺寸必须是块大小的整数倍，src和dst可指向|
    CryptBlocks(dst, src []byte)
}
```

BlockMode接口代表一个工作在块模式（如CBC、ECB等）的加/解密器。

func NewCBCEncrypter

```
func NewCBCEncrypter(b Block, iv []byte) BlockMode
```

返回一个密码分组链接模式的、底层用b加密的BlockMode接口，初始向量iv的长度必须等于b的块尺寸。

Example

```
key := []byte("example key 1234")
plaintext := []byte("exampleplaintext")
// CBC mode works on blocks so plaintexts may need to be padded to
// next whole block. For an example of such padding, see
// https://tools.ietf.org/html/rfc5246#section-6.2.3.2\. Here we'l
// assume that the plaintext is already of the correct length.
if len(plaintext)%aes.BlockSize != 0 {
    panic("plaintext is not a multiple of the block size")
}
block, err := aes.NewCipher(key)
if err != nil {
    panic(err)
}
// The IV needs to be unique, but not secure. Therefore it's commor
// include it at the beginning of the ciphertext.
ciphertext := make([]byte, aes.BlockSize+len(plaintext))
iv := ciphertext[:aes.BlockSize]
if _, err := io.ReadFull(rand.Reader, iv); err != nil {
    panic(err)
}
mode := cipher.NewCBCEncrypter(block, iv)
mode.CryptBlocks(ciphertext[aes.BlockSize:], plaintext)
// It's important to remember that ciphertexts must be authenticate
// (i.e. by using crypto/hmac) as well as being encrypted in order
// be secure.
fmt.Printf("%x\n", ciphertext)
```

func NewCBCDecrypter

```
func NewCBCDecrypter(b Block, iv []byte) BlockMode
```

返回一个密码分组链接模式的、底层用b解密的BlockMode接口，初始向量iv必须和加密时使用的iv相同。

Example

```
key := []byte("example key 1234")
ciphertext, _ := hex.DecodeString("f363f3ccdcb12bb883abf484ba77d9cd
block, err := aes.NewCipher(key)
if err != nil {
    panic(err)
}
// The IV needs to be unique, but not secure. Therefore it's common
// include it at the beginning of the ciphertext.
if len(ciphertext) < aes.BlockSize {
    panic("ciphertext too short")
}
iv := ciphertext[:aes.BlockSize]
ciphertext = ciphertext[aes.BlockSize:]
// CBC mode always works in whole blocks.
if len(ciphertext)%aes.BlockSize != 0 {
    panic("ciphertext is not a multiple of the block size")
}
mode := cipher.NewCBCDecrypter(block, iv)
// CryptBlocks can work in-place if the two arguments are the same.
mode.CryptBlocks(ciphertext, ciphertext)
// If the original plaintext lengths are not a multiple of the bloc
// size, padding would have to be added when encrypting, which woul
// removed at this point. For an example, see
// https://tools.ietf.org/html/rfc5246#section-6.2.3.2\. However, i
// critical to note that ciphertexts must be authenticated (i.e. by
// using crypto/hmac) before being decrypted in order to avoid crea
// a padding oracle.
fmt.Printf("%s\n", ciphertext)
```

Output:

```
exampleplaintext
```

type Stream

```
type Stream interface {
    // 从加密器的key流和src中依次取出字节二者xor后写入dst, src和dst可指向同
    XORKeyStream(dst, src []byte)
}
```

Stream接口代表一个流模式的加/解密器。

func NewCFBEncrypter

```
func NewCFBEncrypter(block Block, iv []byte) Stream
```

返回一个密码反馈模式的、底层用block加密的Stream接口，初始向量iv的长度必须等于block的块尺寸。

Example

```
key := []byte("example key 1234")
plaintext := []byte("some plaintext")
block, err := aes.NewCipher(key)
if err != nil {
    panic(err)
}
// The IV needs to be unique, but not secure. Therefore it's common
// include it at the beginning of the ciphertext.
ciphertext := make([]byte, aes.BlockSize+len(plaintext))
iv := ciphertext[:aes.BlockSize]
if _, err := io.ReadFull(rand.Reader, iv); err != nil {
    panic(err)
}
stream := cipher.NewCFBEncrypter(block, iv)
stream.XORKeyStream(ciphertext[aes.BlockSize:], plaintext)
// It's important to remember that ciphertexts must be authenticate
// (i.e. by using crypto/hmac) as well as being encrypted in order
// be secure.
```

func NewCFBDecrypter

```
func NewCFBDecrypter(block Block, iv []byte) Stream
```

返回一个密码反馈模式的、底层用block解密的Stream接口，初始向量iv必须和加密时使用的iv相同。

Example

```
key := []byte("example key 1234")
ciphertext, _ := hex.DecodeString("22277966616d9bc47177bd02603d08c9
block, err := aes.NewCipher(key)
if err != nil {
    panic(err)
}
// The IV needs to be unique, but not secure. Therefore it's common
// include it at the beginning of the ciphertext.
if len(ciphertext) < aes.BlockSize {
    panic("ciphertext too short")
}
iv := ciphertext[:aes.BlockSize]
ciphertext = ciphertext[aes.BlockSize:]
stream := cipher.NewCFBDecrypter(block, iv)
// XORKeyStream can work in-place if the two arguments are the same
stream.XORKeyStream(ciphertext, ciphertext)
fmt.Printf("%s", ciphertext)
```

Output:

```
some plaintext
```

func NewOFB

```
func NewOFB(b Block, iv []byte) Stream
```

返回一个输出反馈模式的、底层采用b生成key流的Stream接口，初始向量iv的长度必须等于b的块尺寸。

Example

```
key := []byte("example key 1234")
plaintext := []byte("some plaintext")
block, err := aes.NewCipher(key)
if err != nil {
    panic(err)
}
// The IV needs to be unique, but not secure. Therefore it's commor
// include it at the beginning of the ciphertext.
ciphertext := make([]byte, aes.BlockSize+len(plaintext))
iv := ciphertext[:aes.BlockSize]
if _, err := io.ReadFull(rand.Reader, iv); err != nil {
    panic(err)
}
stream := cipher.NewOFB(block, iv)
stream.XORKeyStream(ciphertext[aes.BlockSize:], plaintext)
// It's important to remember that ciphertexts must be authenticate
// (i.e. by using crypto/hmac) as well as being encrypted in order
// be secure.
// OFB mode is the same for both encryption and decryption, so we c
// also decrypt that ciphertext with NewOFB.
plaintext2 := make([]byte, len(plaintext))
stream = cipher.NewOFB(block, iv)
stream.XORKeyStream(plaintext2, ciphertext[aes.BlockSize:])
fmt.Printf("%s\n", plaintext2)
```

Output:

```
some plaintext
```

func NewCTR

```
func NewCTR(block Block, iv []byte) Stream
```

返回一个计数器模式的、底层采用block生成key流的Stream接口，初始向量iv的长度必须等于block的块尺寸。

Example

```
key := []byte("example key 1234")
plaintext := []byte("some plaintext")
block, err := aes.NewCipher(key)
if err != nil {
    panic(err)
}
// The IV needs to be unique, but not secure. Therefore it's commor
// include it at the beginning of the ciphertext.
ciphertext := make([]byte, aes.BlockSize+len(plaintext))
iv := ciphertext[:aes.BlockSize]
if _, err := io.ReadFull(rand.Reader, iv); err != nil {
    panic(err)
}
stream := cipher.NewCTR(block, iv)
stream.XORKeyStream(ciphertext[aes.BlockSize:], plaintext)
// It's important to remember that ciphertexts must be authenticate
// (i.e. by using crypto/hmac) as well as being encrypted in order
// be secure.
// CTR mode is the same for both encryption and decryption, so we c
// also decrypt that ciphertext with NewCTR.
plaintext2 := make([]byte, len(plaintext))
stream = cipher.NewCTR(block, iv)
stream.XORKeyStream(plaintext2, ciphertext[aes.BlockSize:])
fmt.Printf("%s\n", plaintext2)
```

Output:

```
some plaintext
```

type StreamReader

```
type StreamReader struct {
    S   Stream
    R   io.Reader
}
```

将一个Stream与一个io.Reader接口关联起来，Read方法会调用XORKeyStream方法来处理获取的所有切片。

Example

```
key := []byte("example key 1234")
inFile, err := os.Open("encrypted-file")
if err != nil {
    panic(err)
}
defer inFile.Close()
block, err := aes.NewCipher(key)
if err != nil {
    panic(err)
}
// If the key is unique for each ciphertext, then it's ok to use a
// IV.
var iv [aes.BlockSize]byte
stream := cipher.NewOFB(block, iv[:])
outFile, err := os.OpenFile("decrypted-file", os.O_WRONLY|os.O_CREA
if err != nil {
    panic(err)
}
defer outFile.Close()
reader := &cipher.StreamReader{S: stream, R: inFile}
// Copy the input file to the output file, decrypting as we go.
if _, err := io.Copy(outFile, reader); err != nil {
    panic(err)
}
// Note that this example is simplistic in that it omits any
// authentication of the encrypted data. It you were actually to us
// StreamReader in this manner, an attacker could flip arbitrary bi
// the output.
```

func (StreamReader) Read

```
func (r StreamReader) Read(dst []byte) (n int, err error)
```

type StreamWriter

```
type StreamWriter struct {
    S   Stream
    W   io.Writer
    Err error // unused
}
```

将一个Stream与一个io.Writer接口关联起来，Write方法会调用XORKeyStream方法来处理提供的所有切片。如果Write方法返回的n小于提供的切片的长度，则表示StreamWriter不同步，必须丢弃。StreamWriter没有内建的缓存，不需要调用Close

方法去清空缓存。

Example

```
key := []byte("example key 1234")
inFile, err := os.Open("plaintext-file")
if err != nil {
    panic(err)
}
defer inFile.Close()
block, err := aes.NewCipher(key)
if err != nil {
    panic(err)
}
// If the key is unique for each ciphertext, then it's ok to use a
// IV.
var iv [aes.BlockSize]byte
stream := cipher.NewOFB(block, iv[:])
outFile, err := os.OpenFile("encrypted-file", os.O_WRONLY|os.O_CRE
if err != nil {
    panic(err)
}
defer outFile.Close()
writer := &cipher.StreamWriter{S: stream, W: outFile}
// Copy the input file to the output file, encrypting as we go.
if _, err := io.Copy(writer, inFile); err != nil {
    panic(err)
}
// Note that this example is simplistic in that it omits any
// authentication of the encrypted data. It you were actually to us
// StreamReader in this manner, an attacker could flip arbitrary bi
// the decrypted result.
```

func (StreamWriter) Write

```
func (w StreamWriter) Write(src []byte) (n int, err error)
```

func (StreamWriter) Close

```
func (w StreamWriter) Close() error
```

如果w.W字段实现了io.Closer接口，本方法会调用其Close方法并返回该方法的返回值；否则不做操作返回nil。

type AEAD

```
type AEAD interface {
    // 返回提供给Seal和Open方法的随机数nonce的字节长度
    NonceSize() int
    // 返回原始文本和加密文本的最大长度差异
    Overhead() int
    // 加密并认证明文，认证附加的data，将结果添加到dst，返回更新后的切片。
    // nonce的长度必须是NonceSize()字节，且对给定的key和时间都是独一无二的。
    // plaintext和dst可以是同一个切片，也可以不同。
    Seal(dst, nonce, plaintext, data []byte) []byte
    // 解密密文并认证，认证附加的data，如果认证成功，将明文添加到dst，返回更新
    // nonce的长度必须是NonceSize()字节，nonce和data都必须和加密时使用的相
    // ciphertext和dst可以是同一个切片，也可以不同。
    Open(dst, nonce, ciphertext, data []byte) ([]byte, error)
}
```

AEAD接口是一种提供了使用关联数据进行认证加密的功能的加密模式。

func NewGCM

```
func NewGCM(cipher Block) (AEAD, error)
```

函数用迦洛瓦计数器模式包装提供的128位Block接口，并返回AEAD接口。

package des

`import "crypto/des"`

des包实现了DES标准和TDEA算法，参见U.S. Federal Information Processing Standards Publication 46-3。

Index

Constants

```
const BlockSize = 8
```

DES字节块的大小。

type KeySizeError

```
type KeySizeError int
```

func (KeySizeError) Error

```
func (k KeySizeError) Error() string
```

func NewCipher

```
func NewCipher(key []byte) (cipher.Block, error)
```

创建并返回一个使用DES算法的cipher.Block接口。

func NewTripleDESCipher

```
func NewTripleDESCipher(key []byte) (cipher.Block, error)
```

创建并返回一个使用TDEA算法的cipher.Block接口。

Example

```
// NewTripleDESCipher can also be used when EDE2 is required by
// duplicating the first 8 bytes of the 16-byte key.
ede2Key := []byte("example key 1234")
var tripleDESKey []byte
tripleDESKey = append(tripleDESKey, ede2Key[:16]...)
tripleDESKey = append(tripleDESKey, ede2Key[:8]...)
_, err := des.NewTripleDESCipher(tripleDESKey)
if err != nil {
    panic(err)
}
// See crypto/cipher for how to use a cipher.Block for encryption a
// decryption.
```

package dsa

```
import "crypto/dsa"
```

dsa包实现FIPS 186-3定义的数字签名算法（Digital Signature Algorithm），即DSA算法。

Index

Variables

```
var ErrInvalidPublicKey = errors.New("crypto/dsa: invalid public ke
```

非法公钥，FIPS标准的公钥格式是很严格的，但有些实现没这么严格，使用这些实现的公钥时，就会导致这个错误。

type ParameterSizes

```
type ParameterSizes int
```

是DSA参数中的质数可以接受的字位长度的枚举，参见FIPS 186-3 section 4.2。

```
const (
    L1024N160 ParameterSizes = iota
    L2048N224
    L2048N256
    L3072N256
)
```

type Parameters

```
type Parameters struct {
    P, Q, G *big.Int
}
```

Parameters代表密钥的域参数，这些参数可以被一组密钥共享，Q的字位长度必须是8的倍数。

type PublicKey

```
type PublicKey struct {
    Parameters
    Y   *big.Int
}
```

PublicKey代表一个DSA公钥。

type PrivateKey

```
type PrivateKey struct {
    PublicKey
    X   *big.Int
}
```

PrivateKey代表一个DSA私钥。

func GenerateKey

func GenerateParameters

```
func GenerateParameters(params *Parameters, rand io.Reader, sizes F
```

GenerateParameters函数随机设置合法的参数到params。即使机器很快，函数也可能会花费很多时间来生成参数。

```
func GenerateKey(priv *PrivateKey, rand io.Reader) error
```

GenerateKey生成一对公钥和私钥；priv.PublicKey.Parameters字段必须已经（被GenerateParameters函数）设置了合法的参数。

func Sign

```
func Sign(rand io.Reader, priv *PrivateKey, hash []byte) (r, s *big
```

使用私钥对任意长度的hash值（必须是较大信息的hash结果）进行签名，返回签名结果（一对大整数）。私钥的安全性取决于密码读取器的熵度（随机程度）。

注意根据FIPS 186-3 section 4.6的规定，hash必须被截断到亚组的长度，本函数是不会自己截断的。

func Verify

```
func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool
```

使用公钥认证hash和两个大整数r、s构成的签名，报告签名是否合法。

注意根据FIPS 186-3 section 4.6的规定，hash必须被截断到亚组的长度，本函数是不会自己截断的。

package ecdsa

```
import "crypto/ecdsa"
```

ecdsa包实现了椭圆曲线数字签名算法，参见FIPS 186-3。

Index

type PublicKey

```
type PublicKey struct {
    elliptic.Curve
    X, Y *big.Int
}
```

PrivateKey代表一个ECDSA公钥。

type PrivateKey

```
type PrivateKey struct {
    PublicKey
    D   *big.Int
}
```

PrivateKey代表一个ECDSA私钥。

func GenerateKey

GenerateKey函数生成一对

```
func GenerateKey(c elliptic.Curve, rand io.Reader) (priv *PrivateKe
```

公钥/私钥。

func Sign

```
func Sign(rand io.Reader, priv *PrivateKey, hash []byte) (r, s *big
```

使用私钥对任意长度的hash值（必须是较大信息的hash结果）进行签名，返回签名结果（一对大整数）。私钥的安全性取决于密码读取器的熵度（随机程度）。

func Verify

```
func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool
```

使用公钥验证hash值和两个大整数r、s构成的签名，并返回签名是否合法。

package elliptic

```
import "crypto/elliptic"
```

elliptic包实现了几条覆盖素数有限域的标准椭圆曲线。

Index

type Curve

```
type Curve interface {
    // Params返回椭圆曲线的参数
    Params() *CurveParams
    // IsOnCurve判断一个点是否在椭圆曲线上
    IsOnCurve(x, y *big.Int) bool
    // 返回点(x1,y1)和点(x2,y2)相加的结果
    Add(x1, y1, x2, y2 *big.Int) (x, y *big.Int)
    // 返回2*(x,y)，即(x,y)+(x,y)
    Double(x1, y1 *big.Int) (x, y *big.Int)
    // k是一个大端在前格式的数字，返回k*(Bx,By)
    ScalarMult(x1, y1 *big.Int, k []byte) (x, y *big.Int)
    // k是一个大端在前格式的数字，返回k*G，G是本椭圆曲线的基点
    ScalarBaseMult(k []byte) (x, y *big.Int)
}
```

Curve代表一个短格式的Weierstrass椭圆曲线，其中a=-3。

Weierstrass椭圆曲线的格式：y**2 = x**3 + a*x + b

参见http://www.hyperelliptic.org/EFD/g1p/auto-shortw.html

func P224

```
func P224() Curve
```

返回一个实现了P-224的曲线。（参见FIPS 186-3, section D.2.2）

func P256

```
func P256() Curve
```

返回一个实现了P-256的曲线。（参见FIPS 186-3, section D.2.3）

func P384

```
func P384() Curve
```

返回一个实现了P-384的曲线。（参见FIPS 186-3, section D.2.4）

func P521

```
func P521() Curve
```

返回一个实现了P-512的曲线。（参见FIPS 186-3, section D.2.5）

type CurveParams

```
type CurveParams struct {
    P       *big.Int // 决定有限域的p的值（必须是素数）
    N       *big.Int // 基点的阶（必须是素数）
    B       *big.Int // 曲线公式的常量（B!=2）
    Gx, Gy  *big.Int // 基点的坐标
    BitSize int      // 决定有限域的p的字位数
}
```

CurveParams包含一个椭圆曲线的所有参数，也可提供一般的、非常数时间实现的椭圆曲线。

func (*CurveParams) Params

```
func (curve *CurveParams) Params() *CurveParams
```

func (*CurveParams) IsOnCurve

```
func (curve *CurveParams) IsOnCurve(x, y *big.Int) bool
```

func (*CurveParams) Add

```
func (curve *CurveParams) Add(x1, y1, x2, y2 *big.Int) (*big.Int, *
```

func (*CurveParams) Double

```
func (curve *CurveParams) Double(x1, y1 *big.Int) (*big.Int, *big.I
```

func (*CurveParams) ScalarMult

```
func (curve *CurveParams) ScalarMult(Bx, By *big.Int, k []byte) (*b
```

func (*CurveParams) ScalarBaseMult

```
func (curve *CurveParams) ScalarBaseMult(k []byte) (*big.Int, *big.
```

func GenerateKey

```
func GenerateKey(curve Curve, rand io.Reader) (priv []byte, x, y *b
```

GenerateKey返回一个公钥/私钥对。priv是私钥，而(x,y)是公钥。密钥对是通过提供的随机数读取器来生成的，该io.Reader接口必须返回随机数据。

func Marshal

```
func Marshal(curve Curve, x, y *big.Int) []byte
```

Marshal将一个点编码为ANSI X9.62指定的格式。

func Unmarshal

```
func Unmarshal(curve Curve, data []byte) (x, y *big.Int)
```

将一个Marshal编码后的点还原；如果出错，x会被设为nil。

package hmac

```
import "crypto/hmac"
```

hmac包实现了U.S. Federal Information Processing Standards Publication 198规定的HMAC（加密哈希信息认证码）。

HMAC是使用key标记信息的加密hash。接收者使用相同的key逆运算来认证hash。

出于安全目的，接收者应使用Equal函数比较认证码：

```
// 如果messageMAC是message的合法HMAC标签，函数返回真
func CheckMAC(message, messageMAC, key []byte) bool {
    mac := hmac.New(sha256.New, key)
    mac.Write(message)
    expectedMAC := mac.Sum(nil)
    return hmac.Equal(messageMAC, expectedMAC)
}
```

Index

func Equal

```
func Equal(mac1, mac2 []byte) bool
```

比较两个MAC是否相同，而不会泄露对比时间信息。（以规避时间侧信道攻击：指通过计算比较时花费的时间的长短来获取密码的信息，用于密码破解）

func New

```
func New(h func() hash.Hash, key []byte) hash.Hash
```

New函数返回一个采用hash.Hash作为底层hash接口、key作为密钥的HMAC算法的hash接口。

package md5

```
import "crypto/md5"
```

md5包实现了MD5哈希算法，参见RFC 1321。

Index

Examples

Constants

```
const BlockSize = 64
```

MD5字节块大小。

```
const Size = 16
```

MD5校验和字节数。

func Sum

```
func Sum(data []byte) [Size]byte
```

返回数据data的MD5校验和。

Example

```
data := []byte("These pretzels are making me thirsty.")
fmt.Printf("%x", md5.Sum(data))
```

Output:

```
b0804ec967f48520697662a204f5fe72
```

func New

```
func New() hash.Hash
```

返回一个新的使用MD5校验的hash.Hash接口。

Example

```
h := md5.New()
io.WriteString(h, "The fog is getting thicker!")
io.WriteString(h, "And Leon's getting laaarger!")
fmt.Printf("%x", h.Sum(nil))
```

Output:

```
e2c569be17396eca2a2e3c11578123ed
```

package rand

```
import "crypto/rand"
```

rand包实现了用于加解密的更安全的随机数生成器。

Index

Examples

Variables

```
var Reader io.Reader
```

Reader是一个全局、共享的密码用强随机数生成器。在Unix类型系统中，会从/dev/urandom读取；而Windows中会调用CryptGenRandom API。

func Int

```
func Int(rand io.Reader, max *big.Int) (n *big.Int, err error)
```

返回一个在[0, max)区间服从均匀分布的随机值，如果max<=0则会panic。

func Prime

```
func Prime(rand io.Reader, bits int) (p *big.Int, err error)
```

返回一个具有指定字位数的数字，该数字具有很高可能性是质数。如果从rand读取时出错，或者bits<2会返回错误。

func Read

```
func Read(b []byte) (n int, err error)
```

本函数是一个使用io.ReadFull调用Reader.Read的辅助性函数。当且仅当err == nil时，返回值n == len(b)。

Example

```
c := 10
b := make([]byte, c)
_, err := rand.Read(b)
if err != nil {
    fmt.Println("error:", err)
    return
}
// The slice should now contain random bytes instead of only zeroes
fmt.Println(bytes.Equal(b, make([]byte, c)))
```

Output:

```
false
```

package rc4

`import "crypto/rc4"`

rc4包实现了RC4加密算法，参见Bruce Schneier's Applied Cryptography。

Index

type KeySizeError

```
type KeySizeError int
```

func (KeySizeError) Error

```
func (k KeySizeError) Error() string
```

type Cipher

```
type Cipher struct {
    // 内含隐藏或非导出字段
}
```

Cipher是一个使用特定密钥的RC4实例，本类型实现了cipher.Stream接口。

func NewCipher

```
func NewCipher(key []byte) (*Cipher, error)
```

NewCipher创建并返回一个新的Cipher。参数key是RC4密钥，至少1字节，最多256字节。

func (*Cipher) Reset

```
func (c *Cipher) Reset()
```

Reset方法会清空密钥数据，以便将其数据从程序内存中清除（以免被破解）

func (*Cipher) XORKeyStream

```
func (c *Cipher) XORKeyStream(dst, src []byte)
```

XORKeyStream方法将src的数据与秘钥生成的伪随机位流取XOR并写入dst。dst和src可指向同一内存地址；但如果指向不同则其底层内存不可重叠。

Bugs

☞ RC4被广泛使用，但设计上的缺陷使它很少用于较新的协议中。

package rsa

```
import "crypto/rsa"
```

rsa包实现了PKCS#1规定的RSA加密算法。

Index

Constants

```
const (
    // PSSSaltLengthAuto让PSS签名在签名时让盐尽可能长，并在验证时自动检测出
    PSSSaltLengthAuto = 0
    // PSSSaltLengthEqualsHash让盐的长度和用于签名的哈希值的长度相同。
    PSSSaltLengthEqualsHash = -1
)
```

Variables

```
var ErrDecryption = errors.New("crypto/rsa: decryption error")
```

ErrDecryption代表解密数据失败。它故意写的语焉不详，以避免适应性攻击。

```
var ErrMessageTooLong = errors.New("crypto/rsa: message too long fo
```

当试图用公钥加密尺寸过大的数据时，就会返回ErrMessageTooLong。

```
var ErrVerification = errors.New("crypto/rsa: verification error")
```

ErrVerification代表认证签名失败。它故意写的语焉不详，以避免适应性攻击。

type CRTValue

```
type CRTValue struct {
    Exp   *big.Int // D mod (prime-1).
    Coeff *big.Int // R·Coeff ≡ 1 mod Prime.
    R     *big.Int // product of primes prior to this (inc p and q)
}
```

CRTValue包含预先计算的中国剩余定理的值。

type PrecomputedValues

```
type PrecomputedValues struct {
    Dp, Dq *big.Int // D mod (P-1) (or mod Q-1)
    Qinv   *big.Int // Q^-1 mod P
    // CRTValues用于保存第3个及其余的素数的预计算值。
    // 因为历史原因，头两个素数的CRT在PKCS#1中的处理是不同的。
    // 因为互操作性十分重要，我们镜像了这些素数的预计算值。
    CRTValues []CRTValue
}
```

type PublicKey

```
type PublicKey struct {
    N   *big.Int // 模
    E   int      // 公开的指数
}
```

代表一个RSA公钥。

type PrivateKey

```
type PrivateKey struct {
    PublicKey            // 公钥
    D         *big.Int   // 私有的指数
    Primes    []*big.Int // N的素因子，至少有两个
    // 包含预先计算好的值，可在某些情况下加速私钥的操作
    Precomputed PrecomputedValues
}
```

代表一个RSA私钥。

func GenerateKey

```
func GenerateKey(random io.Reader, bits int) (priv *PrivateKey, err
```

GenerateKey函数使用随机数据生成器random生成一对具有指定字位数的RSA密钥。

func GenerateMultiPrimeKey

```
func GenerateMultiPrimeKey(random io.Reader, nprimes int, bits int)
```

GenerateMultiPrimeKey使用指定的字位数生成一对多质数的RSA密钥，参见US patent 4405829。虽然公钥可以和二质数情况下的公钥兼容（事实上，不能区分两种公钥），私钥却不行。因此有可能无法生成特定格式的多质数的密钥对，或不能将生成的密钥用在其他（语言的）代码里。

http://www.cacr.math.uwaterloo.ca/techreports/2006/cacr2006-16.pdf中的Table 1说明了给定字位数的密钥可以接受的质数最大数量。

func (*PrivateKey) Precompute

```
func (priv *PrivateKey) Precompute()
```

Precompute方法会预先进行一些计算，以加速未来的私钥的操作。

func (*PrivateKey) Validate

```
func (priv *PrivateKey) Validate() error
```

Validate方法进行密钥的完整性检查。如果密钥合法会返回nil，否则会返回说明问题的error值。

type PSSOptions

```
type PSSOptions struct {
    // SaltLength控制PSS签名中加盐的长度，可以是字节数，或者某个PSS盐长度的
    SaltLength int
}
```

PSSOptions包含用于创建和认证PSS签名的参数。

func EncryptOAEP

```
func EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey,
```

采用RSA-OAEP算法加密给出的数据。数据不能超过((公共模数的长度)-2*(hash长度)+2)字节。

func DecryptOAEP

```
func DecryptOAEP(hash hash.Hash, random io.Reader, priv *PrivateKey
```

DecryptOAEP解密RSA-OAEP算法加密的数据。如果random不是nil，函数会注意规避时间侧信道攻击。

func EncryptPKCS1v15

```
func EncryptPKCS1v15(rand io.Reader, pub *PublicKey, msg []byte) (c
```

EncryptPKCS1v15使用PKCS#1 v1.5规定的填充方案和RSA算法加密msg。信息不能超过((公共模数的长度)-11)字节。注意：使用本函数加密明文（而不是会话密钥）是危险的，请尽量在新协议中使用RSA OAEP。

func DecryptPKCS1v15

```
func DecryptPKCS1v15(rand io.Reader, priv *PrivateKey, ciphertext [
```

DecryptPKCS1v15使用PKCS#1 v1.5规定的填充方案和RSA算法解密密文。如果random不是nil，函数会注意规避时间侧信道攻击。

func DecryptPKCS1v15SessionKey

```
func DecryptPKCS1v15SessionKey(rand io.Reader, priv *PrivateKey, ci
```

DecryptPKCS1v15SessionKey使用PKCS#1 v1.5规定的填充方案和RSA算法解密会话密钥。如果random不是nil，函数会注意规避时间侧信道攻击。

如果密文长度不对，或者如果密文比公共模数的长度还长，会返回错误；否则，不会返回任何错误。如果填充是合法的，生成的明文信息会拷贝进key；否则，key不会被修改。这些情况都会在固定时间内出现（规避时间侧信道攻击）。本函数的目

的是让程序的使用者事先生成一个随机的会话密钥，并用运行时的值继续协议。这样可以避免任何攻击者从明文窃取信息的可能性。

参见"Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1"。

func SignPKCS1v15

```
func SignPKCS1v15(rand io.Reader, priv *PrivateKey, hash crypto.Has
```

SignPKCS1v15使用RSA PKCS#1 v1.5规定的RSASSA-PKCS1-V1_5-SIGN签名方案计算签名。注意hashed必须是使用提供给本函数的hash参数对（要签名的）原始数据进行hash的结果。

func VerifyPKCS1v15

```
func VerifyPKCS1v15(pub *PublicKey, hash crypto.Hash, hashed []byte
```

VerifyPKCS1v15认证RSA PKCS#1 v1.5签名。hashed是使用提供的hash参数对（要签名的）原始数据进行hash的结果。合法的签名会返回nil，否则表示签名不合法。

func SignPSS

```
func SignPSS(rand io.Reader, priv *PrivateKey, hash crypto.Hash, ha
```

SignPSS采用RSASSA-PSS方案计算签名。注意hashed必须是使用提供给本函数的hash参数对（要签名的）原始数据进行hash的结果。opts参数可以为nil，此时会使用默认参数。

func VerifyPSS

```
func VerifyPSS(pub *PublicKey, hash crypto.Hash, hashed []byte, sig
```

VerifyPSS认证一个PSS签名。hashed是使用提供给本函数的hash参数对（要签名的）原始数据进行hash的结果。合法的签名会返回nil，否则表示签名不合法。opts参数可以为nil，此时会使用默认参数。

package sha1

`import "crypto/sha1"`

sha1包实现了SHA1哈希算法，参见RFC 3174。

Index

Examples

Constants

```
const BlockSize = 64
```

SHA1的块大小。

```
const Size = 20
```

SHA1校验和的字节数。

func Sum

```
func Sum(data []byte) [Size]byte
```

返回数据data的SHA1校验和。

Example

```
data := []byte("This page intentionally left blank.")
fmt.Printf("% x", sha1.Sum(data))
```

Output:

```
af 06 49 23 bb f2 30 15 96 aa c4 c2 73 ba 32 17 8e bc 4a 96
```

func New

```
func New() hash.Hash
```

返回一个新的使用SHA1校验的hash.Hash接口。

Example

```
h := sha1.New()
io.WriteString(h, "His money is twice tainted:")
io.WriteString(h, " 'taint yours and 'taint mine.")
fmt.Printf("% x", h.Sum(nil))
```

Output:

```
59 7f 6a 54 00 10 f9 4c 15 d7 18 06 a9 9a 2c 87 10 e7 47 bd
```

package sha256

`import "crypto/sha256"`

sha256包实现了SHA224和SHA256哈希算法，参见FIPS 180-4。

Index

Constants

```
const BlockSize = 64
```

SHA224和SHA256的字节块大小。

```
const Size = 32
```

SHA256校验和的字节长度。

```
const Size224 = 28
```

SHA224校验和的字节长度。

func Sum256

```
func Sum256(data []byte) [Size]byte
```

返回数据的SHA256校验和。

func New

```
func New() hash.Hash
```

返回一个新的使用SHA256校验算法的hash.Hash接口。

func Sum224

```
func Sum224(data []byte) (sum224 [Size224]byte)
```

返回数据的SHA224校验和。

func New224

```
func New224() hash.Hash
```

返回一个新的使用SHA224校验算法的hash.Hash接口。

package sha512

`import "crypto/sha512"`

sha512包实现了SHA384和SHA512哈希算法，参见FIPS 180-2。

Index

Constants

```
const BlockSize = 128
```

SHA384和SHA512的字节块大小。

```
const Size = 64
```

SHA512校验和的字节长度。

```
const Size384 = 48
```

SHA384校验和的字节长度。

func Sum512

```
func Sum512(data []byte) [Size]byte
```

返回数据的SHA512校验和。

func New

```
func New() hash.Hash
```

返回一个新的使用SHA512校验算法的hash.Hash接口。

func Sum384

```
func Sum384(data []byte) (sum384 [Size384]byte)
```

返回数据的SHA384校验和。

func New384

```
func New384() hash.Hash
```

返回一个新的使用SHA384校验算法的hash.Hash接口。

package subtle

```
import "crypto/subtle"
```

Package subtle implements functions that are often useful in cryptographic code but require careful thought to use correctly.

Index

func ConstantTimeByteEq

```
func ConstantTimeByteEq(x, y uint8) int
```

如果x == y返回1，否则返回0。

func ConstantTimeEq

```
func ConstantTimeEq(x, y int32) int
```

如果x == y返回1，否则返回0。

func ConstantTimeLessOrEq

```
func ConstantTimeLessOrEq(x, y int) int
```

如果x <= y返回1，否则返回0；如果x或y为负数，或者大于2**31-1，函数行为是未定义的。

func ConstantTimeCompare

```
func ConstantTimeCompare(x, y []byte) int
```

如果x、y的长度和内容都相同返回1；否则返回0。消耗的时间正比于切片长度而与内容无关。

func ConstantTimeCopy

```
func ConstantTimeCopy(v int, x, y []byte)
```

如果v == 1,则将y的内容拷贝到x；如果v == 0，x不作修改；其他情况的行为是未定义并应避免的。

func ConstantTimeSelect

```
func ConstantTimeSelect(v, x, y int) int
```

如果v == 1，返回x；如果v == 0，返回y；其他情况的行为是未定义并应避免的。

package tls

```
import "crypto/tls"
```

tls包实现了TLS 1.2，细节参见RFC 5246。

Index

Examples

- Dial

Constants

```
const (
    TLS_RSA_WITH_RC4_128_SHA                uint16 = 0x0005
    TLS_RSA_WITH_3DES_EDE_CBC_SHA           uint16 = 0x000a
    TLS_RSA_WITH_AES_128_CBC_SHA            uint16 = 0x002f
    TLS_RSA_WITH_AES_256_CBC_SHA            uint16 = 0x0035
    TLS_ECDHE_ECDSA_WITH_RC4_128_SHA        uint16 = 0xc007
    TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA    uint16 = 0xc009
    TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA    uint16 = 0xc00a
    TLS_ECDHE_RSA_WITH_RC4_128_SHA          uint16 = 0xc011
    TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA     uint16 = 0xc012
    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA      uint16 = 0xc013
    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA      uint16 = 0xc014
    TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256   uint16 = 0xc02f
    TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 uint16 = 0xc02b
)
```

可选的加密组的ID的列表。参见：http://www.iana.org/assignments/tls-parameters/tls-parameters.xml

```
const (
    VersionSSL30 = 0x0300
    VersionTLS10 = 0x0301
    VersionTLS11 = 0x0302
    VersionTLS12 = 0x0303
)
```

type ClientAuthType

```
type ClientAuthType int
```

ClientAuthType类型声明服务端将遵循的TLS客户端验证策略。

```
const (
    NoClientCert ClientAuthType = iota
    RequestClientCert
    RequireAnyClientCert
    VerifyClientCertIfGiven
    RequireAndVerifyClientCert
)
```

type CurveID

```
type CurveID uint16
```

CurveID是TLS椭圆曲线的标识符的类型。

参见：http://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-8

```
const (
    CurveP256 CurveID = 23
    CurveP384 CurveID = 24
    CurveP521 CurveID = 25
)
```

type Certificate

```
type Certificate struct {
    Certificate [][]byte
    PrivateKey  crypto.PrivateKey // 只支持rsa.PrivateKey和*ecdsa.Pr
    // OCSPStaple包含一个可选的OCSP staple的回复，当客户端要求时用于回复客
    OCSPStaple []byte
    // Leaf是解析后的叶证书，可以使用x509.ParseCertificate初始化，
    // 以简化每次TLS客户端进行客户端认证时握手的过程。如果Leaf是nil，叶证书会
    Leaf *x509.Certificate
}
```

Certificate是一个或多个证书的链条，叶证书在最前面。

func LoadX509KeyPair

```
func LoadX509KeyPair(certFile, keyFile string) (cert Certificate, e
```

LoadX509KeyPair读取并解析一对文件获取公钥和私钥。这些文件必须是PEM编码的。

func X509KeyPair

```
func X509KeyPair(certPEMBlock, keyPEMBlock []byte) (cert Certificat
```

X509KeyPair解析一对PEM编码的数据获取公钥和私钥。

type ClientSessionState

```
type ClientSessionState struct {
    // 内含隐藏或非导出字段
}
```

ClientSessionState包含客户端所需的用于恢复TLS会话的状态。

type ClientSessionCache

```
type ClientSessionCache interface {
    // Get搜索与给出的键相关联的*ClientSessionState并用ok说明是否找到
    Get(sessionKey string) (session *ClientSessionState, ok bool)
    // Put将*ClientSessionState与给出的键关联并写入缓存中
    Put(sessionKey string, cs *ClientSessionState)
}
```

ClientSessionCache是ClientSessionState对象的缓存，可以被客户端用于恢复与某个服务端的TLS会话。本类型的实现期望被不同线程并行的调用。

func NewLRUClientSessionCache

```
func NewLRUClientSessionCache(capacity int) ClientSessionCache
```

函数使用给出的容量创建一个采用LRU策略的ClientSessionState，如果capacity<1会采用默认容量。

type Config

```
type Config struct {
    // Rand提供用于生成随机数和RSA盲签名的熵源，该接口必须能安全的用于并发。
    // 如果Rand是nil，会使用crypto/rand包的密码用随机数读取器。
    Rand io.Reader
    // Time返回当前时间，如果是nil会使用time.Now。
```

```
    Time func() time.Time
    // 不少于一个证书的链，用于提供给连接的另一端，服务端必须保证至少有一个证书
    Certificates []Certificate
    // NameToCertificate映射证书名到证书。
    // 注意证书名可以是"*.example.com "的格式，因此证书名不是必须为域名。
    // 参见Config.BuildNameToCertificate方法。
    // 如本字段为nil，Certificates字段的第一个成员会被用于所有连接。
    NameToCertificate map[string]*Certificate
    // RootCAs定义权威根证书，客户端会在验证服务端证书时用到本字段。
    // 如果RootCAs是nil，TLS会使用主机的根CA池。
    RootCAs *x509.CertPool
    // 可以支持的应用层协议的列表
    NextProtos []string
    // 用于认证返回证书的主机名（除非设置了InsecureSkipVerify）。
    // 也被用在客户端的握手里，以支持虚拟主机。
    ServerName string
    // ClientAuth决定服务端的认证策略，默认是NoClientCert。
    ClientAuth ClientAuthType
    // ClientCAs定义权威根证书，服务端会在采用ClientAuth策略时使用它来认证
    ClientCAs *x509.CertPool
    // InsecureSkipVerify控制客户端是否认证服务端的证书链和主机名。
    // 如果InsecureSkipVerify为真，TLS连接会接受服务端提供的任何证书和该证
    // 此时，TLS连接容易遭受中间人攻击，这种设置只应用于测试。
    InsecureSkipVerify bool
    // CipherSuites是支持的加密组合列表。如果CipherSuites为nil，
    // TLS连接会使用本包的实现支持的密码组合列表。
    CipherSuites []uint16
    // 本字段控制服务端是选择客户端最期望的密码组合还是服务端最期望的密码组合。
    // 如果本字段为真，服务端会优先选择CipherSuites字段中靠前的密码组合使用。
    PreferServerCipherSuites bool
    // SessionTicketsDisabled可以设为假以关闭会话恢复支持。
    SessionTicketsDisabled bool
    // SessionTicketKey被TLS服务端用于提供哦你会话恢复服务，参见RFC 5077。
    // 如果本字段为零值，它会在第一次服务端握手之前填写上随机数据。
    //
    // 如果多个服务端都在终止和同一主机的连接，它们应拥有相同的SessionTicket
    // 如果SessionTicketKey泄露了，使用该键的之前的记录和未来的TLS连接可能
    SessionTicketKey [32]byte
    // SessionCache是ClientSessionState的缓存，用于恢复TLS会话。
    ClientSessionCache ClientSessionCache
    // MinVersion包含可接受的最低SSL/TLS版本。如果为0，会将SSLv3作为最低版
    MinVersion uint16
    // MaxVersion包含可接受的最高SSL/TLS版本。
    // 如果为0，会将本包使用的版本作为最高版本，目前是TLS 1.2。
    MaxVersion uint16
    // 本字段包含用于ECDHE握手的椭圆曲线的ID，按优先度排序。如为空，会使用默认
    CurvePreferences []CurveID
    // 内含隐藏或非导出字段
}
```

Config结构类型用于配置TLS客户端或服务端。在本类型的值提供给TLS函数后，就不应再修改该值。Config类型值可能被重用；tls包也不会修改它。

func (*Config) BuildNameToCertificate

```
func (c *Config) BuildNameToCertificate()
```

BuildNameToCertificate解析c.Certificates并将每一个叶证书的CommonName和SubjectAlternateName字段用于创建c.NameToCertificate。

type ConnectionState

```
type ConnectionState struct {
    Version                    uint16                // 连接使用的TL
    HandshakeComplete          bool                  // TLS握手是否完
    DidResume                  bool                  // 连接恢复了之前
    CipherSuite                uint16                // 使用的加密程序
    NegotiatedProtocol         string                // 商定的下一层协
    NegotiatedProtocolIsMutual bool                  // 商定的协议是服
    ServerName                 string                // 服务端名（仅服
    PeerCertificates           []*x509.Certificate   // 远端提供的证书
    VerifiedChains             [][]*x509.Certificate // 从PeerCerti
}
```

ConnectionState类型记录连接的基本TLS细节。

type Conn

```
type Conn struct {
    // 内含隐藏或非导出字段
}
```

Conn代表一个安全连接。本类型实现了net.Conn接口。

func (*Conn) LocalAddr

```
func (c *Conn) LocalAddr() net.Addr
```

LocalAddr返回本地网络地址。

func (*Conn) RemoteAddr

```
func (c *Conn) RemoteAddr() net.Addr
```

LocalAddr返回远端网络地址。

func (*Conn) ConnectionState

```
func (c *Conn) ConnectionState() ConnectionState
```

ConnectionState返回该连接的基本TLS细节。

func (*Conn) SetDeadline

```
func (c *Conn) SetDeadline(t time.Time) error
```

SetDeadline设置该连接的读写操作绝对期限。t为Time零值表示不设置超时。在一次Write/Read方法超时后，TLS连接状态会被破坏，之后所有的读写操作都会返回同一错误。

func (*Conn) SetReadDeadline

```
func (c *Conn) SetReadDeadline(t time.Time) error
```

SetReadDeadline设置该连接的读操作绝对期限。t为Time零值表示不设置超时。

func (*Conn) SetWriteDeadline

```
func (c *Conn) SetWriteDeadline(t time.Time) error
```

SetReadDeadline设置该连接的写操作绝对期限。t为Time零值表示不设置超时。在一次Write方法超时后，TLS连接状态会被破坏，之后所有的写操作都会返回同一错误。

func (*Conn) Handshake

```
func (c *Conn) Handshake() error
```

Handshake执行客户端或服务端的握手协议（如果还没有执行的话）。本包的大多数应用不需要显式的调用Handsake方法：第一次Read或Write方法会自动调用本方法。

func (*Conn) VerifyHostname

```
func (c *Conn) VerifyHostname(host string) error
```

VerifyHostname检查用于连接到host的对等实体证书链是否合法。如果合法，它会返回nil；否则，会返回一个描述该问题的错误。

func (*Conn) OCSPResponse

```
func (c *Conn) OCSPResponse() []byte
```

OCSPResponse返回来自服务端的OCSP staple回复（如果有）。只有客户端可以使用本方法。

func (*Conn) Read

```
func (c *Conn) Read(b []byte) (n int, err error)
```

Read从连接读取数据，可设置超时，参见SetDeadline和SetReadDeadline。

func (*Conn) Write

```
func (c *Conn) Write(b []byte) (int, error)
```

Write将数据写入连接，可设置超时，参见SetDeadline和SetWriteDeadline。

func (*Conn) Close

```
func (c *Conn) Close() error
```

Close关闭连接。

func Client

```
func Client(conn net.Conn, config *Config) *Conn
```

Client使用conn作为下层传输接口返回一个TLS连接的客户端侧。配置参数config必须是非nil的且必须设置了ServerName或者InsecureSkipVerify字段。

func Server

```
func Server(conn net.Conn, config *Config) *Conn
```

Server使用conn作为下层传输接口返回一个TLS连接的服务端侧。配置参数config必须是非nil的且必须含有至少一个证书。

func Dial

```
func Dial(network, addr string, config *Config) (*Conn, error)
```

Dial使用net.Dial连接指定的网络和地址，然后发起TLS握手，返回生成的TLS连接。Dial会将nil的配置视为零值的配置；参见Config类型的文档获取细节。

Example

```
// Connecting with a custom root-certificate set.
const rootPEM = `
-----BEGIN CERTIFICATE-----
MIIEBDCCAuygAwIBAgIDAjppMA0GCSqGSIb3DQEBBQUAMEIxCzAJBgNVBAYTAlVT
MRYwFAYDVQQKEw1HZW9UcnVzdCBJbmMuMRswGQYDVQQDExJHZW9UcnVzdCBHbG9i
YWwgQ0EwHhcNMTMwNDA1MTUxNTU1WhcNMTUwNDA0MTUxNTU1WjBJMQswCQYDVQQG
EwJVUzETMBEGA1UEChMKR29vZ2xlIEluYzElMCMGA1UEAxMcR29vZ2xlIEludGVy
bmV0IEF1dGhvcml0eSBHMjCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEB
AJwqBHdc2FCROgajguDYUEi8iT/xGXAaiEZ+4I/F8YnOIe5a/mENtzJEiaB0C1NP
VaTOgmKV7utZX8bhBYASxF6UP7xbSDj0U/ck5vuR6RXEz/RTDfRK/J9U3n2+oGtv
h8DQUB8oMANA2ghzUWx//zo8pzcGjr1LEQTrfSTe5vn8MXH7lNVg8y5Kr0LSy+rE
ahqyzFPdFUuLH8gZYR/Nnag+YyuENWllhMgZxUYi+FOVvuOAShDGKuy6lyARxzmZ
EASg8GF6lSWMTlJ14rbtCMoU/M4iarNOz0YDl5cDfsCx3nuvRTPPuj5xt970JSXC
DTWJnZ37DhF5iR43xa+OcmkCAwEAAaOB+zCB+DAfBgNVHSMEGDAWgBTAephojYn7
qwVkDBF9qn1luMrMTjAdBgNVHQ4EFgQUSt0GFhu89mi1dvWBtrtiGrpagS8wEgYD
VR0TAQH/BAgwBgEB/wIBADAOBgNVHQ8BAf8EBAMCAQYwOgYDVR0fBDMwMTAvoC2g
K4YpaHR0cDovL2NybC5nZW90cnVzdC5jb20vY3Jscy9ndGdsb2JhbC5jcmwwPQYI
KwYBBQUHAQEEMTAvMC0GCCsGAQUFBzABhiFodHRwOi8vZ3RnbG9iYWwtb2NzcC5n
ZW90cnVzdC5jb20wFwYDVR0gBBAwDjAMBgorBgEEAdZ5AgUBMA0GCSqGSIb3DQEB
BQUAA4IBAQA21waAESetKhSbOHezI6B1WLuxfoNCunLaHtiONgaX4PCVOzf9G0JY
/iLIa704XtE7JW4S615ndkZAkNoUyHgN7ZVm2o6Gb4ChulYylYbc3GrKBIxbf/a/
zG+FA1jDaFETzf3I93k9mTXwVqO94FntT0QJo544evZG0R0SnU++0ED8Vf4GXjza
HFa9llF7b1cq26KqltyMdMKVvvBulRP/F/A8rLIQjcxz++iPAsbw+zOzlTvjwsto
WHPbqCRiOwY1nQ2pM714A5AuTHhdUDqB1O6gyHA43LL5Z/qHQF1hwFGPa4NrzQU6
yuGnBXj8ytqU0CwIPX4WecigUCAkVDNx
-----END CERTIFICATE-----`
// First, create the set of root certificates. For this example we
// have one. It's also possible to omit this in order to use the
// default root set of the current operating system.
roots := x509.NewCertPool()
ok := roots.AppendCertsFromPEM([]byte(rootPEM))
if !ok {
    panic("failed to parse root certificate")
}
conn, err := tls.Dial("tcp", "mail.google.com:443", &tls.Config{
    RootCAs: roots,
})
if err != nil {
    panic("failed to connect: " + err.Error())
}
conn.Close()
```

func DialWithDialer

```
func DialWithDialer(dialer *net.Dialer, network, addr string, conf:
```

DialWithDialer使用dialer.Dial连接指定的网络和地址，然后发起TLS握手，返回生成的TLS连接。dialer中的超时和期限设置会将连接和TLS握手作为一个整体来应用。

DialWithDialer会将nil的配置视为零值的配置；参见Config类型的文档获取细节。

func Listen

```
func Listen(network, laddr string, config *Config) (net.Listener, e
```

函数创建一个TLS监听器，使用net.Listen函数接收给定地址上的连接。配置参数config必须是非nil的且必须含有至少一个证书。

func NewListener

```
func NewListener(inner net.Listener, config *Config) net.Listener
```

函数创建一个TLS监听器，该监听器接受inner接收到的每一个连接，并调用Server函数包装这些连接。配置参数config必须是非nil的且必须含有至少一个证书。

package x509

```
import "crypto/x509"
```

x509包解析X.509编码的证书和密钥。

Index

Examples

Constants

```
const (
    PEMCipherDES
    PEMCipher3DES
    PEMCipherAES128
    PEMCipherAES192
    PEMCipherAES256
)
```

可能会被EncryptPEMBlock加密算法使用的值。

Variables

```
var ErrUnsupportedAlgorithm = errors.New("x509: cannot verify signa
```

当试图执行包含目前未实现的算法的操作时，会返回ErrUnsupportedAlgorithm。

```
var IncorrectPasswordError = errors.New("x509: decryption password
```

当检测到不正确的密码时，会返回IncorrectPasswordError。

type PEMCipher

```
type PEMCipher int
```

type PublicKeyAlgorithm

```
type PublicKeyAlgorithm int
```

```
const (
    UnknownPublicKeyAlgorithm PublicKeyAlgorithm = iota
    RSA
    DSA
    ECDSA
)
```

type SignatureAlgorithm

```
type SignatureAlgorithm int
```

```
const (
    UnknownSignatureAlgorithm SignatureAlgorithm = iota
    MD2WithRSA
    MD5WithRSA
    SHA1WithRSA
    SHA256WithRSA
    SHA384WithRSA
    SHA512WithRSA
    DSAWithSHA1
    DSAWithSHA256
    ECDSAWithSHA1
    ECDSAWithSHA256
    ECDSAWithSHA384
    ECDSAWithSHA512
)
```

type SystemRootsError

```
type SystemRootsError struct {
}
```

当从系统装载根证书失败时，会返回SystemRootsError。

func (SystemRootsError) Error

```
func (e SystemRootsError) Error() string
```

type HostnameError

```
type HostnameError struct {
    Certificate *Certificate
    Host        string
}
```

当认证的名字和请求的名字不匹配时，会返回HostnameError。

func (HostnameError) Error

```
func (h HostnameError) Error() string
```

type UnknownAuthorityError

```
type UnknownAuthorityError struct {
    // 内含隐藏或非导出字段
}
```

当证书的发布者未知时，会返回UnknownAuthorityError。

func (UnknownAuthorityError) Error

```
func (e UnknownAuthorityError) Error() string
```

type ConstraintViolationError

```
type ConstraintViolationError struct{}
```

当请求的用途不被证书许可时，会返回ConstraintViolationError。如：当公钥不是证书的签名密钥时用它检查签名。

func (ConstraintViolationError) Error

```
func (ConstraintViolationError) Error() string
```

type UnhandledCriticalExtension

```
type UnhandledCriticalExtension struct{}
```

func (UnhandledCriticalExtension) Error

```
func (h UnhandledCriticalExtension) Error() string
```

type CertificateInvalidError

```
type CertificateInvalidError struct {
    Cert   *Certificate
    Reason InvalidReason
}
```

当发生其余的错误时，会返回CertificateInvalidError。本包的使用者可能会想统一处理所有这类错误。

func (CertificateInvalidError) Error

```
func (e CertificateInvalidError) Error() string
```

type KeyUsage

```
type KeyUsage int
```

KeyUsage代表给定密钥的合法操作集。用KeyUsage类型常数的位图表示。（字位表示有无）

```
const (
    KeyUsageDigitalSignature KeyUsage = 1 << iota
    KeyUsageContentCommitment
    KeyUsageKeyEncipherment
    KeyUsageDataEncipherment
    KeyUsageKeyAgreement
    KeyUsageCertSign
    KeyUsageCRLSign
    KeyUsageEncipherOnly
    KeyUsageDecipherOnly
)
```

type ExtKeyUsage

```
type ExtKeyUsage int
```

ExtKeyUsage代表给定密钥的合法操作扩展集。每一个ExtKeyUsage类型常数定义一个特定的操作。

```
const (
    ExtKeyUsageAny ExtKeyUsage = iota
    ExtKeyUsageServerAuth
    ExtKeyUsageClientAuth
    ExtKeyUsageCodeSigning
    ExtKeyUsageEmailProtection
    ExtKeyUsageIPSECEndSystem
    ExtKeyUsageIPSECTunnel
    ExtKeyUsageIPSECUser
    ExtKeyUsageTimeStamping
    ExtKeyUsageOCSPSigning
    ExtKeyUsageMicrosoftServerGatedCrypto
    ExtKeyUsageNetscapeServerGatedCrypto
)
```

type VerifyOptions

```
type VerifyOptions struct {
    DNSName       string
    Intermediates *CertPool
    Roots         *CertPool // 如为nil，将使用系统根证书池
    CurrentTime   time.Time // 如为零值，将使用当前时间
    // KeyUsage指定了可以接受哪些密钥扩展用途，空列表代表ExtKeyUsageServe
    // 密钥用途被作为生成证书链的限制条件（类似Windows加密应用程序接口的行为，
    // 要接受任何密钥用途，可以使本字段包含ExtKeyUsageAny。
    KeyUsages []ExtKeyUsage
}
```

VerifyOptions包含提供给Certificate.Verify方法的参数。它是结构体类型，因为其他PKIX认证API需要很长参数。

type InvalidReason

```
type InvalidReason int
```

```
const (
    // NotAuthorizedToSign表示给本证书签名的证书不是CA证书
    NotAuthorizedToSign InvalidReason = iota
    // Expired表示证书已过期，根据VerifyOptions.CurrentTime判断
    Expired
    // CANotAuthorizedForThisName表示中间证书或根证书具有名字限制，且不包
    CANotAuthorizedForThisName
    // TooManyIntermediates表示违反了路径长度限制
    TooManyIntermediates
    // IncompatibleUsage表示证书的密钥用途显示它只能用于其它目的
    IncompatibleUsage
)
```

type Certificate

```
type Certificate struct {
    Raw                     []byte // 原始、完整的ASN.1 DER内容（证书、
    RawTBSCertificate       []byte // ASN.1 DER 内容的证书部分
    RawSubjectPublicKeyInfo []byte // 原始DER编码的SubjectPublicKeyI
    RawSubject              []byte // 原始DER编码的Subject
    RawIssuer               []byte // 原始DER编码的Issuer
    Signature          []byte
    SignatureAlgorithm SignatureAlgorithm
    PublicKeyAlgorithm PublicKeyAlgorithm
    PublicKey          interface{}
    Version             int
    SerialNumber        *big.Int
    Issuer              pkix.Name
    Subject             pkix.Name
    NotBefore, NotAfter time.Time // 有效期前后界，本时间段之外无效
    KeyUsage            KeyUsage
    // Extensions保管原始的X.509扩展。当解析证书时，本字段用于摘录本包未解析
    // 序列化证书时，Extensions字段会被忽略，参见ExtraExtensions。
    Extensions []pkix.Extension
    // ExtraExtensions包含应被直接拷贝到任何序列化的证书中的扩展。
    // 本字段保管的值会覆盖任何其它字段生成的扩展。
    // ExtraExtensions字段在解析证书时不会被填写，参见Extensions。
    ExtraExtensions []pkix.Extension
    ExtKeyUsage        []ExtKeyUsage           // 密钥扩展用途的序列
    UnknownExtKeyUsage []asn1.ObjectIdentifier // 遇到的本包不能识别的
    BasicConstraintsValid bool // 如果下两个字段合法，将为真
    IsCA                  bool
    MaxPathLen            int
    SubjectKeyId   []byte
    AuthorityKeyId []byte
    // RFC 5280, 4.2.2.1（认证信息存取）
    OCSPServer            []string
    IssuingCertificateURL []string
    // 证书持有者的替用名称
    DNSNames       []string
    EmailAddresses []string
    IPAddresses    []net.IP
    // 名称的约束
    PermittedDNSDomainsCritical bool // 如为真则名称约束被标记为关键的
    PermittedDNSDomains         []string
    // CRL配销点
    CRLDistributionPoints []string
    PolicyIdentifiers []asn1.ObjectIdentifier
}
```

Certificate代表一个X.509证书。

func (*Certificate) CheckSignatureFrom

```
func (c *Certificate) CheckSignatureFrom(parent *Certificate) (err
```

CheckSignatureFrom检查c中的签名是否是来自parent的合法签名。

func (*Certificate) CheckCRLSignature

```
func (c *Certificate) CheckCRLSignature(crl *pkix.CertificateList)
```

CheckCRLSignature检查crl中的签名是否来自c。

func (*Certificate) CheckSignature

```
func (c *Certificate) CheckSignature(algo SignatureAlgorithm, signe
```

CheckSignature检查signature是否是c的公钥生成的signed的合法签名。

func (*Certificate) CreateCRL

```
func (c *Certificate) CreateCRL(rand io.Reader, priv interface{}, r
```

CreateCRL返回一个DER编码的CRL（证书注销列表），使用c签名，并包含给出的已取消签名列表。

只支持RSA类型的密钥（priv参数必须是*rsa.PrivateKey类型）。

func (*Certificate) Equal

```
func (c *Certificate) Equal(other *Certificate) bool
```

func (*Certificate) Verify

```
func (c *Certificate) Verify(opts VerifyOptions) (chains [][]*Certi
```

Verify通过创建一到多个从c到opts.Roots中的证书的链条来认证c，如有必要会使用opts.Intermediates中的证书。如果成功，它会返回一到多个证书链条，每一条都以c开始，以opts.Roots中的证书结束。

警告：它不会做任何取消检查。

Example

```
// Verifying with a custom list of root certificates.
const rootPEM = `
-----BEGIN CERTIFICATE-----
MIIEBDCCAuygAwIBAgIDAjppMA0GCSqGSIb3DQEBBQUAMEIxCzAJBgNVBAYTAlVT
MRYwFAYDVQQKEw1HZW9UcnVzdCBJbmMuMRswGQYDVQQDExJHZW9UcnVzdCBHbG9i
YWwgQ0EwHhcNMTMwNDA1MTUxNTU1WhcNMTUwNDA0MTUxNTU1WjBJMQswCQYDVQQG
EwJVUzETMBEGA1UEChMKR29vZ2xlIEluYzElMCMGA1UEAxMcR29vZ2xlIEludGVy
bmV0IEF1dGhvcml0eSBHMjCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEB
AJwqBHdc2FCROgajguDYUEi8iT/xGXAaiEZ+4I/F8YnOIe5a/mENtzJEiaB0C1NP
VaTOgmKV7utZX8bhBYASxF6UP7xbSDj0U/ck5vuR6RXEz/RTDfRK/J9U3n2+oGtv
h8DQUB8oMANA2ghzUWx//zo8pzcGjr1LEQTrfSTe5vn8MXH7lNVg8y5Kr0LSy+rE
ahqyzFPdFUuLH8gZYR/Nnag+YyuENWllhMgZxUYi+FOVvuOAShDGKuy6lyARxzmZ
EASg8GF6lSWMTlJ14rbtCMoU/M4iarNOz0YDl5cDfsCx3nuvRTPPuj5xt970JSXC
DTWJnZ37DhF5iR43xa+OcmkCAwEAAaOB+zCB+DAfBgNVHSMEGDAWgBTAephojYn7
qwVkDBF9qn1luMrMTjAdBgNVHQ4EFgQUSt0GFhu89mi1dvWBtrtiGrpagS8wEgYD
VR0TAQH/BAgwBgEB/wIBADAOBgNVHQ8BAf8EBAMCAQYwOgYDVR0fBDMwMTAvoC2g
K4YpaHR0cDovL2NybC5nZW90cnVzdC5jb20vY3Jscy9ndGdsb2JhbC5jcmwwPQYI
KwYBBQUHAQEEMTAvMC0GCCsGAQUFBzABhiFodHRwOi8vZ3RnbG9iYWwtb2NzcC5n
ZW90cnVzdC5jb20wFwYDVR0gBBAwDjAMBgorBgEEAdZ5AgUBMA0GCSqGSIb3DQEB
BQUAA4IBAQA21waAESetKhSbOHezI6B1WLuxfoNCunLaHtiONgaX4PCVOzf9G0JY
/iLIa704XtE7JW4S615ndkZAkNoUyHgN7ZVm2o6Gb4ChulYylYbc3GrKBIxbf/a/
zG+FA1jDaFETzf3I93k9mTXwVqO94FntT0QJo544evZG0R0SnU++0ED8Vf4GXjza
HFa9llF7b1cq26KqltyMdMKVvvBulRP/F/A8rLIQjcxz++iPAsbw+zOzlTvjwsto
WHPbqCRiOwY1nQ2pM714A5AuTHhdUDqB1O6gyHA43LL5Z/qHQF1hwFGPa4NrzQU6
yuGnBXj8ytqU0CwIPX4WecigUCAkVDNx
-----END CERTIFICATE-----`
const certPEM = `
-----BEGIN CERTIFICATE-----
MIIDujCCAqKgAwIBAgIIE31FZVaPXTUwDQYJKoZIhvcNAQEFBQAwSTELMAkGA1UE
BhMCVVMxEzARBgNVBAoTCkdvb2dsZSBJbmMxJTAjBgNVBAMTHEdvb2dsZSBJbnRl
cm5ldCBBdXRob3JpdHkgRzIwHhcNMTQwMTI5MTMyNzQzWhcNMTQwNTI5MDAwMDAw
WjBpMQswCQYDVQQGEwJVUzETMBEGA1UECAwKQ2FsaWZvcm5pYTEWMBQGA1UEBwwN
TW91bnRhaW4gVmlldzETMBEGA1UECgwKR29vZ2xlIEluYzEYMBYGA1UEAwwPbWFp
bC5nb29nbGUuY29tMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEfRrObuSW5T7q
5CnSEqefEmtH4CCv6+5EckuriNr1CjfVvqzwfAhopXkLrq45EQm8vkmf7W96XJhC
7ZM0dYi1/qOCAU8wggFLMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjAa
BgNVHREEEzARgg9tYWlsLmdvb2dsZS5jb20wCwYDVR0PBAQDAgeAMGgGCCsGAQUF
BwEBBFwwWjArBggrBgEFBQcwAoYfaHR0cDovL3BraS5nb29nbGUuY29tL0dJQUcy
LmNydDArBggrBgEFBQcwAYYfaHR0cDovL2NsaWVudHMxLmdvb2dsZS5jb20vb2Nz
cDAdBgNVHQ4EFgQUiJxtimAuTfwb+aUtBn5UYKreKvMwDAYDVR0TAQH/BAIwADAf
BgNVHSMEGDAWgBRK3QYWG7z2aLV29YG2u2IaulqBLzAXBgNVHSAEEDAOMAwGCisG
AQQB1nkCBQEwMAYDVR0fBCkwJzAloCOgIYYfaHR0cDovL3BraS5nb29nbGUuY29t
L0dJQUcyLmNybDANBgkqhkiG9w0BAQUFAAOCAQEAH6RYHxHdcGpMpFE3oxDoFnP+
gtuBCHan2yE2GRbJ2Cw8Lw0MmuKqHlf9RSeYfd3BXeKkj1qO6TVKwCh+0HdZk283
```

```
TZZyzmEOyclm3UGFYe82P/iDFt+CeQ3NpmBg+GoaVCuWAARJN/KfglbLyyYygcQq
0SgeDh8dRKUiaW3HQSoYvTvdTuqzwK4CXsr3b5/dAOY8uMuG/IAR3FgwTbZ1dtoW
RvOTa8hYiU6A475WuZKyEHcwnGYe57u2I2KbMgcKjPniocj4QzgYsVAVKW3IwaOh
yE+vPxsiUkvQHdO2fojCkY8jg70jxM+gu59tPDNbw3Uh/2Ij310FgTHsnGQMyA==
-----END CERTIFICATE-----`
// First, create the set of root certificates. For this example we
// have one. It's also possible to omit this in order to use the
// default root set of the current operating system.
roots := x509.NewCertPool()
ok := roots.AppendCertsFromPEM([]byte(rootPEM))
if !ok {
    panic("failed to parse root certificate")
}
block, _ := pem.Decode([]byte(certPEM))
if block == nil {
    panic("failed to parse certificate PEM")
}
cert, err := x509.ParseCertificate(block.Bytes)
if err != nil {
    panic("failed to parse certificate: " + err.Error())
}
opts := x509.VerifyOptions{
    DNSName: "mail.google.com",
    Roots:   roots,
}
if _, err := cert.Verify(opts); err != nil {
    panic("failed to verify certificate: " + err.Error())
}
```

func (*Certificate) VerifyHostname

```
func (c *Certificate) VerifyHostname(h string) error
```

如果c是名为h的主机的合法证书，VerifyHostname会返回真；否则它返回一个描述该不匹配情况的错误。

type CertPool

```
type CertPool struct {
    // 内含隐藏或非导出字段
}
```

CertPool代表一个证书集合/证书池。

func NewCertPool

```
func NewCertPool() *CertPool
```

NewCertPool创建一个新的、空的CertPool。

func (*CertPool) AddCert

```
func (s *CertPool) AddCert(cert *Certificate)
```

AddCert向s中添加一个证书。

func (*CertPool) AppendCertsFromPEM

```
func (s *CertPool) AppendCertsFromPEM(pemCerts []byte) (ok bool)
```

AppendCertsFromPEM试图解析一系列PEM编码的证书。它将找到的任何证书都加入s中，如果所有证书都成功被解析，会返回真。

在许多Linux系统中，/etc/ssl/cert.pem会包含适合本函数的大量系统级根证书。

func (*CertPool) Subjects

```
func (s *CertPool) Subjects() (res [][]byte)
```

Subjects返回池中所有证书的DER编码的持有者的列表。

type CertificateRequest

```
type CertificateRequest struct {
    Raw                      []byte // 原始、完整的ASN.1 DER内容（CSR、
    RawTBSCertificateRequest []byte // ASN.1 DER 内容的证书请求信息
    RawSubjectPublicKeyInfo  []byte // 原始DER编码的SubjectPublicKey
    RawSubject               []byte // 原始DER编码的Subject
    Version            int
    Signature          []byte
    SignatureAlgorithm SignatureAlgorithm
    PublicKeyAlgorithm PublicKeyAlgorithm
    PublicKey          interface{}
    Subject pkix.Name
    // Attributes提供关于证书持有者的额外信息，参见RFC 2986 section 4.1。
    Attributes []pkix.AttributeTypeAndValueSET
    // Extensions保管原始的X.509扩展。当解析CSR时，本字段用于摘录本包未解析
    Extensions []pkix.Extension
    // ExtraExtensions包含应被直接拷贝到任何序列化的CSR中的扩展。
    // 本字段保管的值会覆盖任何其它字段生成的扩展，但会被Attributes字段指定的
    // ExtraExtensions字段在解析CSR时不会增加，参见Extensions。
    ExtraExtensions []pkix.Extension
    // 证书持有者的替用名称。
    DNSNames       []string
    EmailAddresses []string
    IPAddresses    []net.IP
}
```

CertificateRequest代表一个PKCS #10证书签名请求。

func MarshalECPrivateKey

```
func MarshalECPrivateKey(key *ecdsa.PrivateKey) ([]byte, error)
```

MarshalECPrivateKey将ecdsa私钥序列化为ASN.1 DER编码。

func MarshalPKCS1PrivateKey

```
func MarshalPKCS1PrivateKey(key *rsa.PrivateKey) []byte
```

MarshalPKCS1PrivateKey将rsa私钥序列化为ASN.1 PKCS#1 DER编码。

func MarshalPKIXPublicKey

```
func MarshalPKIXPublicKey(pub interface{}) ([]byte, error)
```

MarshalPKIXPublicKey将公钥序列化为PKIX格式DER编码。

func ParseECPrivateKey

```
func ParseECPrivateKey(der []byte) (key *ecdsa.PrivateKey, err erro
```

ParseECPrivateKey解析ASN.1 DER编码的ecdsa私钥。

func ParsePKCS1PrivateKey

```
func ParsePKCS1PrivateKey(der []byte) (key *rsa.PrivateKey, err err
```

ParsePKCS1PrivateKey解析ASN.1 PKCS#1 DER编码的rsa私钥。

func ParsePKCS8PrivateKey

```
func ParsePKCS8PrivateKey(der []byte) (key interface{}, err error)
```

ParsePKCS8PrivateKey解析一个未加密的PKCS#8私钥，参见
http://www.rsa.com/rsalabs/node.asp?id=2130和RFC5208。

func ParsePKIXPublicKey

```
func ParsePKIXPublicKey(derBytes []byte) (pub interface{}, err erro
```

ParsePKIXPublicKey解析一个DER编码的公钥。这些公钥一般在以"BEGIN PUBLIC KEY"出现的PEM块中。

func EncryptPEMBlock

```
func EncryptPEMBlock(rand io.Reader, blockType string, data, passwo
```

EncryptPEMBlock使用指定的密码、加密算法加密data，返回一个具有指定块类型，保管加密后数据的PEM块。

func IsEncryptedPEMBlock

```
func IsEncryptedPEMBlock(b *pem.Block) bool
```

IsEncryptedPEMBlock返回PEM块b是否是用密码加密了的。

func DecryptPEMBlock

```
func DecryptPEMBlock(b *pem.Block, password []byte) ([]byte, error)
```

DecryptPEMBlock接受一个加密后的PEM块和加密该块的密码password，返回解密后的DER编码字节切片。它会检查DEK信息头域，以确定用于解密的算法。如果b中没有DEK信息头域，会返回错误。如果函数发现密码不正确，会返回IncorrectPasswordError。

func ParseCRL

```
func ParseCRL(crlBytes []byte) (certList *pkix.CertificateList, er
```

ParseCRL从crlBytes中解析CRL（证书注销列表）。因为经常有PEM编码的CRL出现在应该是DER编码的地方，因此本函数可以透明的处理PEM编码，只要没有前导的垃圾数据。

func ParseDERCRL

```
func ParseDERCRL(derBytes []byte) (certList *pkix.CertificateList,
```

ParseDERCRL从derBytes中解析DER编码的CRL。

func ParseCertificate

```
func ParseCertificate(asn1Data []byte) (*Certificate, error)
```

ParseCertificate从ASN.1 DER数据解析单个证书。

func ParseCertificateRequest

```
func ParseCertificateRequest(asn1Data []byte) (*CertificateRequest,
```

ParseCertificateRequest解析一个ASN.1 DER数据获取单个证书请求。

func ParseCertificates

```
func ParseCertificates(asn1Data []byte) ([]*Certificate, error)
```

ParseCertificates从ASN.1 DER编码的asn1Data中解析一到多个证书。这些证书必须是串联的，且中间没有填充。

func CreateCertificate

```
func CreateCertificate(rand io.Reader, template, parent *Certificat
```

CreateCertificate基于模板创建一个新的证书。会用到模板的如下字段：

SerialNumber、Subject、NotBefore、NotAfter、KeyUsage、ExtKeyUsage、UnknownExtKeyUsage、

BasicConstraintsValid、IsCA、MaxPathLen、SubjectKeyId、DNSNames、PermittedDNSDomainsCritical、

PermittedDNSDomains、SignatureAlgorithm。

该证书会使用parent签名。如果parent和template相同，则证书是自签名的。Pub参数是被签名者的公钥，而priv是签名者的私钥。

返回的切片是DER编码的证书。

只支持RSA和ECDSA类型的密钥。（pub可以是*rsa.PublicKey*或ecdsa.PublicKey，priv可以是*rsa.PrivateKey*或ecdsa.PrivateKey）

func CreateCertificateRequest

```
func CreateCertificateRequest(rand io.Reader, template *Certificate
```

CreateCertificateRequest基于模板创建一个新的证书请求。会用到模板的如下字段：

Subject、Attributes、Extension、SignatureAlgorithm、DNSNames、EmailAddresses、IPAddresses。

priv是签名者的私钥。返回的切片是DER编码的证书请求。

只支持RSA（*rsa.PrivateKey*）和*ECDSA*（ecdsa.PrivateKey）类型的密钥。

package pkix

```
import "crypto/x509/pkix"
```

pkix包提供了共享的、低层次的结构体，用于ASN.1解析和X.509证书、CRL、OCSP的序列化。

Index

type Extension

```
type Extension struct {
    Id       asn1.ObjectIdentifier
    Critical bool `asn1:"optional"`
    Value    []byte
}
```

Extension代表一个同名的ASN.1结构体，参见RFC 5280, section 4.2。

type AlgorithmIdentifier

```
type AlgorithmIdentifier struct {
    Algorithm  asn1.ObjectIdentifier
    Parameters asn1.RawValue `asn1:"optional"`
}
```

AlgorithmIdentifier代表一个同名的ASN.1结构体，参见RFC 5280, section 4.1.1.2。

type RevokedCertificate

```
type RevokedCertificate struct {
    SerialNumber   *big.Int
    RevocationTime time.Time
    Extensions     []Extension `asn1:"optional"`
}
```

RevokedCertificate代表一个同名的ASN.1结构体，参见RFC 5280, section 5.1。

type TBSCertificateList

```
type TBSCertificateList struct {
    Raw                 asn1.RawContent
    Version             int `asn1:"optional,default:2"`
    Signature           AlgorithmIdentifier
    Issuer              RDNSequence
    ThisUpdate          time.Time
    NextUpdate          time.Time
    RevokedCertificates []RevokedCertificate `asn1:"optional"`
    Extensions          []Extension          `asn1:"tag:0,optional,
}
```

TBSCertificateList代表一个同名的ASN.1结构体，参见RFC 5280, section 5.1。

type AttributeTypeAndValue

```
type AttributeTypeAndValue struct {
    Type  asn1.ObjectIdentifier
    Value interface{}
}
```

AttributeTypeAndValue代表一个同名的ASN.1结构体，参见http://tools.ietf.org/html/rfc5280#section-4.1.2.4。

type AttributeTypeAndValueSET

```
type AttributeTypeAndValueSET struct {
    Type  asn1.ObjectIdentifier
    Value [][]AttributeTypeAndValue `asn1:"set"`
}
```

AttributeTypeAndValueSET代表AttributeTypeAndValue序列表示的ASN.1序列的集合，参见RFC 2986 (PKCS #10)。

type CertificateList

```
type CertificateList struct {
    TBSCertList        TBSCertificateList
    SignatureAlgorithm AlgorithmIdentifier
    SignatureValue     asn1.BitString
}
```

CertificateList代表一个同名的ASN.1结构体，参见RFC 5280, section 5.1。用于认证签名。

func (*CertificateList) HasExpired

```
func (certList *CertificateList) HasExpired(now time.Time) bool
```

HasExpired报告证书列表是否已过期。

type RelativeDistinguishedNameSET

```
type RelativeDistinguishedNameSET []AttributeTypeAndValue
```

type RDNSequence

```
type RDNSequence []RelativeDistinguishedNameSET
```

type Name

```
type Name struct {
    Country, Organization, OrganizationalUnit []string
    Locality, Province                        []string
    StreetAddress, PostalCode                 []string
    SerialNumber, CommonName                  string
    Names []AttributeTypeAndValue
}
```

Name代表一个X.509识别名。只包含识别名的公共属性，额外的属性被忽略。

func (*Name) FillFromRDNSequence

```
func (n *Name) FillFromRDNSequence(rdns *RDNSequence)
```

func (Name) ToRDNSequence

```
func (n Name) ToRDNSequence() (ret RDNSequence)
```

package database

package sql

```
import "database/sql"
```

sql包提供了保证SQL或类SQL数据库的泛用接口。

使用sql包时必须注入（至少）一个数据库驱动。参见http://golang.org/s/sqldrivers 获取驱动列表。

更多用法示例，参见wiki页面：http://golang.org/s/sqlwiki。

Index

Examples

Variables

```
var ErrNoRows = errors.New("sql: no rows in result set")
```

当QueryRow方法没有返回一个row时，调用返回值的Scan方法会返回ErrNoRows。此时，QueryRow返回一个占位的*Row值，延迟本错误直到调用Scan方法时才释放。

```
var ErrTxDone = errors.New("sql: Transaction has already been comm
```

func Register

```
func Register(name string, driver driver.Driver)
```

Register注册并命名一个数据库，可以在Open函数中使用该命名启用该驱动。如果Register注册同一名称两次，或者driver参数为nil，会导致panic。

type Scanner

```
type Scanner interface {
    // Scan方法从数据库驱动获取一个值。
    //
    // 参数src的类型保证为如下类型之一：
    //
    //    int64
    //    float64
    //    bool
    //    []byte
    //    string
    //    time.Time
    //    nil - 表示NULL值
    //
    // 如果不能不丢失信息的保存一个值，应返回错误。
    Scan(src interface{}) error
}
```

Scanner接口会被Rows或Row的Scan方法使用。

type NullBool

```
type NullBool struct {
    Bool  bool
    Valid bool // 如果Bool不是NULL则Valid为真
}
```

NullBool代表一个可为NULL的布尔值。NullBool实现了Scanner接口，因此可以作为Rows/Row的Scan方法的参数保存扫描结果，类似NullString。

func (*NullBool) Scan

```
func (n *NullBool) Scan(value interface{}) error
```

Scan实现了Scanner接口。

func (NullBool) Value

```
func (n NullBool) Value() (driver.Value, error)
```

Value实现了driver.Valuer接口。

type NullInt64

```
type NullInt64 struct {
    Int64 int64
    Valid bool // 如果Int64不是NULL则Valid为真
}
```

NullInt64代表一个可为NULL的int64值。NullInt64实现了Scanner接口，因此可以作为Rows/Row的Scan方法的参数保存扫描结果，类似NullString。

func (*NullInt64) Scan

```
func (n *NullInt64) Scan(value interface{}) error
```

Scan实现了Scanner接口。

func (NullInt64) Value

```
func (n NullInt64) Value() (driver.Value, error)
```

Value实现了driver.Valuer接口。

type NullFloat64

```
type NullFloat64 struct {
    Float64 float64
    Valid   bool // 如果Float64不是NULL则Valid为真
}
```

NullFloat64代表一个可为NULL的float64值。NullFloat64实现了Scanner接口，因此可以作为Rows/Row的Scan方法的参数保存扫描结果，类似NullString。

func (*NullFloat64) Scan

```
func (n *NullFloat64) Scan(value interface{}) error
```

Scan实现了Scanner接口。

func (NullFloat64) Value

```
func (n NullFloat64) Value() (driver.Value, error)
```

Value实现了driver.Valuer接口。

type NullString

```
type NullString struct {
    String string
    Valid  bool // 如果String不是NULL则Valid为真
}
```

NullString代表一个可为NULL的字符串。NullString实现了Scanner接口，因此可以作为Rows/Row的Scan方法的参数保存扫描结果：

```
var s NullString
err := db.QueryRow("SELECT name FROM foo WHERE id=?", id).Scan(&s)
...
if s.Valid {
   // use s.String
} else {
   // NULL value
}
```

func (*NullString) Scan

```
func (ns *NullString) Scan(value interface{}) error
```

Scan实现了Scanner接口。

func (NullString) Value

```
func (ns NullString) Value() (driver.Value, error)
```

Value实现了driver.Valuer接口。

type RawBytes

```
type RawBytes []byte
```

RawBytes是一个字节切片，保管对内存的引用，为数据库自身所使用。在Scaner接口的Scan方法写入RawBytes数据后，该切片只在限次调用Next、Scan或Close方法之前合法。

type Result

```
type Result interface {
    // LastInsertId返回一个数据库生成的回应命令的整数。
    // 当插入新行时，一般来自一个"自增"列。
    // 不是所有的数据库都支持该功能，该状态的语法也各有不同。
    LastInsertId() (int64, error)

    // RowsAffected返回被update、insert或delete命令影响的行数。
    // 不是所有的数据库都支持该功能。
    RowsAffected() (int64, error)
}
```

Result是对已执行的SQL命令的总结。

type DB

```
type DB struct {
    // 内含隐藏或非导出字段
}
```

DB是一个数据库（操作）句柄，代表一个具有零到多个底层连接的连接池。它可以安全的被多个go程同时使用。

sql包会自动创建和释放连接；它也会维护一个闲置连接的连接池。如果数据库具有单连接状态的概念，该状态只有在事务中被观察时才可信。一旦调用了BD.Begin，返回的Tx会绑定到单个连接。当调用事务Tx的Commit或Rollback后，该事务使用的连接会归还到DB的闲置连接池中。连接池的大小可以用SetMaxIdleConns方法控制。

func Open

```
func Open(driverName, dataSourceName string) (*DB, error)
```

Open打开一个dirverName指定的数据库，dataSourceName指定数据源，一般包至少括数据库文件名和（可能的）连接信息。

大多数用户会通过数据库特定的连接帮助函数打开数据库，返回一个*DB。Go标准库中没有数据库驱动。参见http://golang.org/s/sqldrivers获取第三方驱动。

Open函数可能只是验证其参数，而不创建与数据库的连接。如果要检查数据源的名称是否合法，应调用返回值的Ping方法。

返回的DB可以安全的被多个go程同时使用，并会维护自身的闲置连接池。这样一来，Open函数只需调用一次。很少需要关闭DB。

func (*DB) Driver

```
func (db *DB) Driver() driver.Driver
```

Driver方法返回数据库下层驱动。

func (*DB) Ping

```
func (db *DB) Ping() error
```

Ping检查与数据库的连接是否仍有效，如果需要会创建连接。

func (*DB) Close

```
func (db *DB) Close() error
```

Close关闭数据库，释放任何打开的资源。一般不会关闭DB，因为DB句柄通常被多个go程共享，并长期活跃。

func (*DB) SetMaxOpenConns

```
func (db *DB) SetMaxOpenConns(n int)
```

SetMaxOpenConns设置与数据库建立连接的最大数目。

如果n大于0且小于最大闲置连接数，会将最大闲置连接数减小到匹配最大开启连接数的限制。

如果n <= 0，不会限制最大开启连接数，默认为0（无限制）。

func (*DB) SetMaxIdleConns

```
func (db *DB) SetMaxIdleConns(n int)
```

SetMaxIdleConns设置连接池中的最大闲置连接数。

如果n大于最大开启连接数，则新的最大闲置连接数会减小到匹配最大开启连接数的限制。

如果n <= 0，不会保留闲置连接。

func (*DB) Exec

```
func (db *DB) Exec(query string, args ...interface{}) (Result, erro
```

Exec执行一次命令（包括查询、删除、更新、插入等），不返回任何执行结果。参数args表示query中的占位参数。

func (*DB) Query

```
func (db *DB) Query(query string, args ...interface{}) (*Rows, erro
```

Query执行一次查询，返回多行结果（即Rows），一般用于执行select命令。参数args表示query中的占位参数。

Example

```
age := 27
rows, err := db.Query("SELECT name FROM users WHERE age=?", age)
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
for rows.Next() {
    var name string
    if err := rows.Scan(&name); err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s is %d\n", name, age)
}
if err := rows.Err(); err != nil {
    log.Fatal(err)
}
```

func (*DB) QueryRow

```
func (db *DB) QueryRow(query string, args ...interface{}) *Row
```

QueryRow执行一次查询，并期望返回最多一行结果（即Row）。QueryRow总是返回非nil的值，直到返回值的Scan方法被调用时，才会返回被延迟的错误。（如：未找到结果）

Example

```
id := 123
var username string
err := db.QueryRow("SELECT username FROM users WHERE id=?", id).Sca
switch {
case err == sql.ErrNoRows:
    log.Printf("No user with that ID.")
case err != nil:
    log.Fatal(err)
default:
    fmt.Printf("Username is %s\n", username)
}
```

func (*DB) Prepare

```
func (db *DB) Prepare(query string) (*Stmt, error)
```

Prepare创建一个准备好的状态用于之后的查询和命令。返回值可以同时执行多个查询和命令。

func (*DB) Begin

```
func (db *DB) Begin() (*Tx, error)
```

Begin开始一个事务。隔离水平由数据库驱动决定。

type Row

```
type Row struct {
    // 内含隐藏或非导出字段
}
```

QueryRow方法返回Row，代表单行查询结果。

func (*Row) Scan

```
func (r *Row) Scan(dest ...interface{}) error
```

Scan将该行查询结果各列分别保存进dest参数指定的值中。如果该查询匹配多行，Scan会使用第一行结果并丢弃其余各行。如果没有匹配查询的行，Scan会返回ErrNoRows。

type Rows

```
type Rows struct {
    // 内含隐藏或非导出字段
}
```

Rows是查询的结果。它的游标指向结果集的第零行，使用Next方法来遍历各行结果：

```
rows, err := db.Query("SELECT ...")
...
defer rows.Close()
for rows.Next() {
    var id int
    var name string
    err = rows.Scan(&id, &name)
    ...
}
err = rows.Err() // get any error encountered during iteration
...
```

func (*Rows) Columns

```
func (rs *Rows) Columns() ([]string, error)
```

Columns返回列名。如果Rows已经关闭会返回错误。

func (*Rows) Scan

```
func (rs *Rows) Scan(dest ...interface{}) error
```

Scan将当前行各列结果填充进dest指定的各个值中。

如果某个参数的类型为*[]byte*，*Scan*会保存对应数据的拷贝，该拷贝为调用者所有，可以安全的,修改或无限期的保存。如果参数类型为\RawBytes可以避免拷贝；参见RawBytes的文档获取其使用的约束。

如果某个参数的类型为*interface{}，Scan会不做转换的拷贝底层驱动提供的值。如果值的类型为[]byte，会进行数据的拷贝，调用者可以安全使用该值。

func (*Rows) Next

```
func (rs *Rows) Next() bool
```

Next准备用于Scan方法的下一行结果。如果成功会返回真，如果没有下一行或者出现错误会返回假。Err应该被调用以区分这两种情况。

每一次调用Scan方法，甚至包括第一次调用该方法，都必须在前面先调用Next方法。

func (*Rows) Close

```
func (rs *Rows) Close() error
```

Close关闭Rows，阻止对其更多的列举。 如果Next方法返回假，Rows会自动关闭，满足。检查Err方法结果的条件。Close方法时幂等的（多次调用无效的成功），不影响Err方法的结果。

func (*Rows) Err

```
func (rs *Rows) Err() error
```

Err返回可能的、在迭代时出现的错误。Err需在显式或隐式调用Close方法后调用。

type Stmt

```
type Stmt struct {
    // 内含隐藏或非导出字段
}
```

Stmt是准备好的状态。Stmt可以安全的被多个go程同时使用。

func (*Stmt) Exec

```
func (s *Stmt) Exec(args ...interface{}) (Result, error)
```

Exec使用提供的参数执行准备好的命令状态，返回Result类型的该状态执行结果的总结。

func (*Stmt) Query

```
func (s *Stmt) Query(args ...interface{}) (*Rows, error)
```

Query使用提供的参数执行准备好的查询状态，返回Rows类型查询结果。

func (*Stmt) QueryRow

```
func (s *Stmt) QueryRow(args ...interface{}) *Row
```

QueryRow使用提供的参数执行准备好的查询状态。如果在执行时遇到了错误，该错误会被延迟，直到返回值的Scan方法被调用时才释放。返回值总是非nil的。如果没有查询到结果，*Row类型返回值的Scan方法会返回ErrNoRows；否则，Scan方法会扫描结果第一行并丢弃其余行。

示例用法：

```
var name string
err := nameByUseridStmt.QueryRow(id).Scan(&name)
```

func (*Stmt) Close

```
func (s *Stmt) Close() error
```

Close关闭状态。

type Tx

```
type Tx struct {
    // 内含隐藏或非导出字段
}
```

Tx代表一个进行中的数据库事务。

一次事务必须以对Commit或Rollback的调用结束。

调用Commit或Rollback后，所有对事务的操作都会失败并返回错误值ErrTxDone。

func (*Tx) Exec

```
func (tx *Tx) Exec(query string, args ...interface{}) (Result, erro
```

Exec执行命令，但不返回结果。例如执行insert和update。

func (*Tx) Query

```
func (tx *Tx) Query(query string, args ...interface{}) (*Rows, erro
```

Query执行查询并返回零到多行结果（Rows），一般执行select命令。

func (*Tx) QueryRow

```
func (tx *Tx) QueryRow(query string, args ...interface{}) *Row
```

QueryRow执行查询并期望返回最多一行结果（Row）。QueryRow总是返回非nil的结果，查询失败的错误会延迟到在调用该结果的Scan方法时释放。

func (*Tx) Prepare

```
func (tx *Tx) Prepare(query string) (*Stmt, error)
```

Prepare准备一个专用于该事务的状态。返回的该事务专属状态操作在Tx递交会回滚后不能再使用。要在该事务中使用已存在的状态，参见Tx.Stmt方法。

func (*Tx) Stmt

```
func (tx *Tx) Stmt(stmt *Stmt) *Stmt
```

Stmt使用已存在的状态生成一个该事务特定的状态。

示例：

```
updateMoney, err := db.Prepare("UPDATE balance SET money=money+? WH
...
tx, err := db.Begin()
...
res, err := tx.Stmt(updateMoney).Exec(123.45, 98293203)
```

func (*Tx) Commit

```
func (tx *Tx) Commit() error
```

Commit递交事务。

func (*Tx) Rollback

```
func (tx *Tx) Rollback() error
```

Rollback放弃并回滚事务。

package driver

`import "database/sql/driver"`

driver包定义了应被数据库驱动实现的接口，这些接口会被sql包使用。

绝大多数代码应使用sql包。

Index

Variables

```
var Bool boolType
```

Bool是ValueConverter接口值，用于将输入的值转换为布尔类型。

转换规则如下：

```
- 布尔类型：不做修改
- 整数类型：
     1 为真
     0 为假
     其余整数会导致错误
- 字符串和[]byte：与strconv.ParseBool的规则相同
- 所有其他类型都会导致错误
```

```
var Int32 int32Type
```

Int32是一个ValueConverter接口值，用于将值转换为int64类型，会尊重int32类型的限制。

```
var String stringType
```

String是一个ValueConverter接口值，用于将值转换为字符串。如果值v是字符串或者[]byte类型，不会做修改，如果值v是其它类型，会转换为fmt.Sprintf("%v", v)。

```
var DefaultParameterConverter defaultConverter
```

DefaultParameterConverter是ValueConverter接口的默认实现，当一个Stmt没有实现ColumnConverter时，就会使用它。

如果值value满足函数IsValue(value)为真，DefaultParameterConverter直接返回value。否则，整数类型会被转换为int64，浮点数转换为float64，字符串转换为[]byte。其它类型会导致错误。

```
var ResultNoRows noRows
```

ResultNoRows是预定义的Result类型值，用于当一个DDL命令（如create table）成功时被驱动返回。它的LastInsertId和RowsAffected方法都返回错误。

```
var ErrBadConn = errors.New("driver: bad connection")
```

ErrBadConn应被驱动返回，以通知sql包一个driver.Conn处于损坏状态（如服务端之前关闭了连接），sql包会重启一个新的连接。

为了避免重复的操作，如果数据库服务端执行了操作，就不应返回ErrBadConn。即使服务端返回了一个错误。

```
var ErrSkip = errors.New("driver: skip fast-path; continue as if ur
```

ErrSkip可能会被某些可选接口的方法返回，用于在运行时表明快速方法不可用，sql包应像未实现该接口的情况一样执行。ErrSkip只有文档显式说明的地方才支持。

type Value

```
type Value interface{}
```

Value是驱动必须能处理的值。它要么是nil，要么是如下类型的实例：

```
int64
float64
bool
[]byte
string    [*] Rows.Next不会返回该类型值
time.Time
```

type Valuer

```
type Valuer interface {
    // Value返回一个驱动支持的Value类型值
    Value() (Value, error)
}
```

Valuer是提供Value方法的接口。实现了Valuer接口的类型可以将自身转换为驱动支持的Value类型值。

func IsValue

```
func IsValue(v interface{}) bool
```

IsValue报告v是否是合法的Value类型参数。和IsScanValue不同，IsValue接受字符串类型。

func IsScanValue

```
func IsScanValue(v interface{}) bool
```

IsScanValue报告v是否是合法的Value扫描类型参数。和IsValue不同，IsScanValue不接受字符串类型。

type ValueConverter

```
type ValueConverter interface {
    // ConvertValue将一个值转换为驱动支持的Value类型
    ConvertValue(v interface{}) (Value, error)
}
```

ValueConverter接口提供了ConvertValue方法。

driver包提供了各种ValueConverter接口的实现，以保证不同驱动之间的实现和转换的一致性。ValueConverter接口有如下用途：

```
* 转换sql包提供的Value类型值到数据库指定列的类型，并保证它的匹配，
  例如保证某个int64值满足一个表的uint16列。

* 转换数据库提供的值到驱动的Value类型。

* 在扫描时被sql包用于将驱动的Value类型转换为用户的类型。
```

type ColumnConverter

```
type ColumnConverter interface {
    // ColumnConverter返回指定列的ValueConverter
    // 如果该列未指定类型，或不应特殊处理，应返回DefaultValueConverter
    ColumnConverter(idx int) ValueConverter
}
```

如果Stmt有自己的列类型，可以实现ColumnConverter接口，返回值可以将任意类型转换为驱动的Value类型。

type NotNull

```
type NotNull struct {
    Converter ValueConverter
}
```

NotNull实现了ValueConverter接口，不允许nil值，否则会将值交给Converter字段处理。

func (NotNull) ConvertValue

```
func (n NotNull) ConvertValue(v interface{}) (Value, error)
```

type Null

```
type Null struct {
    Converter ValueConverter
}
```

Null实现了ValueConverter接口，允许nil值，否则会将值交给Converter字段处理。

func (Null) ConvertValue

```
func (n Null) ConvertValue(v interface{}) (Value, error)
```

type Driver

```
type Driver interface {
    // Open返回一个新的与数据库的连接，参数name的格式是驱动特定的。
    //
    // Open可能返回一个缓存的连接（之前关闭的连接），但这么做是不必要的；
    // sql包会维护闲置连接池以便有效的重用连接。
    //
    // 返回的连接同一时间只会被一个go程使用。
    Open(name string) (Conn, error)
}
```

Driver接口必须被数据库驱动实现。

type Conn

```
type Conn interface {
    // Prepare返回一个准备好的、绑定到该连接的状态。
    Prepare(query string) (Stmt, error)

    // Close作废并停止任何现在准备好的状态和事务，将该连接标注为不再使用。
    //
    // 因为sql包维护着一个连接池，只有当闲置连接过剩时才会调用Close方法，
    // 驱动的实现中不需要添加自己的连接缓存池。
    Close() error

    // Begin开始并返回一个新的事务。
    Begin() (Tx, error)
}
```

Conn是与数据库的连接。该连接不会被多线程并行使用。连接被假定为具有状态的。

type Execer

```
type Execer interface {
    Exec(query string, args []Value) (Result, error)
}
```

Execer是一个可选的、可能被Conn接口实现的接口。

如果一个Conn未实现Execer接口，sql包的DB.Exec会首先准备一个查询，执行状态，然后关闭状态。Exec可能会返回ErrSkip。

type Queryer

```
type Queryer interface {
    Query(query string, args []Value) (Rows, error)
}
```

Queryer是一个可选的、可能被Conn接口实现的接口。

如果一个Conn未实现Queryer接口，sql包的DB.Query会首先准备一个查询，执行状态，然后关闭状态。Query可能会返回ErrSkip。

type Stmt

```
type Stmt interface {
    // Close关闭Stmt。
    //
    // 和Go1.1一样，如果Stmt被任何查询使用中的话，将不会被关闭。
    Close() error

    // NumInput返回占位参数的个数。
    //
    // 如果NumInput返回值 >= 0，sql包会提前检查调用者提供的参数个数，
    // 并且会在调用Exec或Query方法前返回数目不对的错误。
    //
    // NumInput可以返回-1，如果驱动占位参数的数量。
    // 此时sql包不会提前检查参数个数。
    NumInput() int

    // Exec执行查询，而不会返回结果，如insert或update。
    Exec(args []Value) (Result, error)

    // Query执行查询并返回结果，如select。
    Query(args []Value) (Rows, error)
}
```

Stmt是准备好的状态。它会绑定到一个连接，不应被多go程同时使用。

type Tx

```
type Tx interface {
    Commit() error
    Rollback() error
}
```

Tx是一次事务。

type Result

```
type Result interface {
    // LastInsertId返回insert等命令后数据库自动生成的ID
    LastInsertId() (int64, error)

    // RowsAffected返回被查询影响的行数
    RowsAffected() (int64, error)
}
```

Result是查询执行的结果。

type RowsAffected

```
type RowsAffected int64
```

RowsAffected实现了Result接口，用于insert或update操作，这些操作会修改零到多行数据。

func (RowsAffected) LastInsertId

```
func (RowsAffected) LastInsertId() (int64, error)
```

func (RowsAffected) RowsAffected

```
func (v RowsAffected) RowsAffected() (int64, error)
```

type Rows

```
type Rows interface {
    // Columns返回各列的名称，列的数量可以从切片长度确定。
    // 如果某个列的名称未知，对应的条目应为空字符串。
    Columns() []string

    // Close关闭Rows。
    Close() error

    // 调用Next方法以将下一行数据填充进提供的切片中。
    // 提供的切片必须和Columns返回的切片长度相同。
    //
    // 切片dest可能被填充同一种驱动Value类型，但字符串除外。
    // 所有string值都必须转换为[]byte。
    //
    // 当没有更多行时，Next应返回io.EOF。
    Next(dest []Value) error
}
```

Rows是执行查询得到的结果的迭代器。

package encoding

```
import "encoding"
```

encoding包定义了供其它包使用的可以将数据在字节水平和文本表示之间转换的接口。encoding/gob、encoding/json、encoding/xml三个包都会检查使用这些接口。因此，只要实现了这些接口一次，就可以在多个包里使用。标准包内建类型time.Time和net.IP都实现了这些接口。接口是成对的，分别产生和还原编码后的数据。

Index

type BinaryMarshaler

```
type BinaryMarshaler interface {
    MarshalBinary() (data []byte, err error)
}
```

实现了BinaryMarshaler接口的类型可以将自身序列化为binary格式。

type BinaryUnmarshaler

```
type BinaryUnmarshaler interface {
    UnmarshalBinary(data []byte) error
}
```

实现了BinaryUnmarshaler接口的类型可以将binary格式表示的自身解序列化。

UnmarshalBinary必须可以解码MarshalBinary生成的binary格式数据。本函数可能会对data内容作出修改，所以如果要保持data的数据请事先进行拷贝。

type TextMarshaler

```
type TextMarshaler interface {
    MarshalText() (text []byte, err error)
}
```

实现了BinaryMarshaler接口的类型可以将自身序列化为utf-8编码的textual格式。

type TextUnmarshaler

```
type TextUnmarshaler interface {
    UnmarshalText(text []byte) error
}
```

实现了TextUnmarshaler接口的类型可以将textual格式表示的自身解序列化。

UnmarshalText必须可以解码MarshalText生成的textual格式数据。本函数可能会对data内容作出修改，所以如果要保持data的数据请事先进行拷贝。

package ascii85

```
import "encoding/ascii85"
```

ascii85包实现了ascii85数据编码（5个ascii字符表示4个字节），该编码用于btoa工具和Adobe的PostScript语言和PDF文档格式。

Index

type CorruptInputError

```
type CorruptInputError int64
```

func (CorruptInputError) Error

```
func (e CorruptInputError) Error() string
```

func MaxEncodedLen

```
func MaxEncodedLen(n int) int
```

返回n字节源数据编码后的最大字节数。

func Encode

```
func Encode(dst, src []byte) int
```

将src编码成最多MaxEncodedLen(len(src))数据写入dst，返回实际写入的字节数。编码每4字节一段进行一次，最后一个片段采用特殊的处理方式，因此不应将本函数用于处理大数据流的某一独立数据块。

一般来说ascii85编码数据会被'<~'和'~>'包括起来，函数并未添加上它们。

func Decode

```
func Decode(dst, src []byte, flush bool) (ndst, nsrc int, err erro
```

将src解码后写入dst，返回写入dst的字节数、从src解码的字节数。如果src含有非法数据，函数将返回成功执行的数据（两个数字）和CorruptInputError。如果flush为真，则函数会认为src代表输入流的结尾，完全处理src，而不会等待另一个32字节的数据块。

函数会忽略src中的空格和控制字符，一般来说ascii85编码数据会被'<~'和'~>'包括起来，但是调用者应自行去掉它们。

func NewEncoder

```
func NewEncoder(w io.Writer) io.WriteCloser
```

创建一个将数据编码为ascii85流写入w的编码器。Ascii85编码算法操作32位块，写入结束后，必须调用Close方法将缓存中保留的不完整块刷新到w里。

func NewDecoder

```
func NewDecoder(r io.Reader) io.Reader
```

创建一个从r解码ascii85流的解码器。

package asn1

```
import "encoding/asn1"
```

asn1包实现了DER编码的ASN.1数据结构的解析，参见ITU-T Rec X.690。

其他细节参见"A Layman's Guide to a Subset of ASN.1, BER, and DER"。

网址http://luca.ntop.org/Teaching/Appunti/asn1.html

Index

type SyntaxError

```
type SyntaxError struct {
    Msg string
}
```

SyntaxErrorLeixing表示ASN.1数据不合法。

func (SyntaxError) Error

```
func (e SyntaxError) Error() string
```

type StructuralError

```
type StructuralError struct {
    Msg string
}
```

StructuralError表示ASN.1数据合法但接收的Go类型不匹配。

func (StructuralError) Error

```
func (e StructuralError) Error() string
```

type RawContent

```
type RawContent []byte
```

RawContent用于标记未解码的应被结构体保留的DER数据。如要使用它，结构体的第一个字段必须是本类型，其它字段不能是本类型。

type RawValue

```
type RawValue struct {
    Class, Tag int
    IsCompound bool
    Bytes      []byte
    FullBytes  []byte // 包括标签和长度
}
```

RawValue代表一个未解码的ASN.1对象。

type Flag

```
type Flag bool
```

Flag接收任何数据，如果数据存在就设自身为真。

type Enumerated

```
type Enumerated int
```

Enumerated表示一个明文整数。

type BitString

```
type BitString struct {
    Bytes     []byte // 字位流打包在字节流里
    BitLength int    // 字位流的长度
}
```

BitString类型是用于表示ASN.1 BIT STRING类型的结构体。字位流补齐到最近的字节数保存在内存里并记录合法字位数，补齐的位可以为0个。

func (BitString) At

```
func (b BitString) At(i int) int
```

At方法发挥index位置的字位，如果index出界则返回0。

func (BitString) RightAlign

```
func (b BitString) RightAlign() []byte
```

RightAlign方法返回b表示的字位流的右对齐版本（即补位在开始部分）切片，该切片可能和b共享底层内存。

type ObjectIdentifier

```
type ObjectIdentifier []int
```

ObjectIdentifier类型用于表示ASN.1 OBJECT IDENTIFIER类型。

func (ObjectIdentifier) Equal

```
func (oi ObjectIdentifier) Equal(other ObjectIdentifier) bool
```

如果oi和other代表同一个标识符，Equal方法返回真。

func (ObjectIdentifier) String

```
func (oi ObjectIdentifier) String() string
```

func Marshal

```
func Marshal(val interface{}) ([]byte, error)
```

Marshal函数返回val的ASN.1编码。

此外还提供了供Unmarshal函数识别的结构体标签，可用如下标签：

```
ia5:          使字符串序列化为ASN.1 IA5String类型
omitempty:    使空切片被跳过
printable:    使字符串序列化为ASN.1 PrintableString类型
utf8:         使字符串序列化为ASN.1 UTF8字符串
```

func Unmarshal

```
func Unmarshal(b []byte, val interface{}) (rest []byte, err error)
```

Unmarshal函数解析DER编码的ASN.1结构体数据并使用reflect包填写val指向的任意类型值。因为本函数使用了reflect包，结构体必须使用大写字母起始的字段名。

ASN.1 INTEGER 类型值可以写入int、int32、int64或*big.Int（math/big包）类型。类型不匹配会返回解析错误。

ASN.1 BIT STRING类型值可以写入BitString类型。

ASN.1 OCTET STRING类型值可以写入[]byte类型。

ASN.1 OBJECT IDENTIFIER类型值可以写入ObjectIdentifier类型。

ASN.1 ENUMERATED类型值可以写入Enumerated类型。

ASN.1 UTCTIME类型值或GENERALIZEDTIME 类型值可以写入time.Time类型。

ASN.1 PrintableString类型值或者IA5String类型值可以写入string类型。

以上任一ASN.1类型值都可写入interface{}类型。保存在接口里的类型为对应的Go类型，ASN.1整型对应int64。

如果类型x可以写入切片的成员类型，则类型x的ASN.1 SEQUENCE或SET类型可以写入该切片。

ASN.1 SEQUENCE或SET类型如果其每一个成员都可以写入某结构体的对应字段，则可以写入该结构体

对Unmarshal函数，下列字段标签有特殊含义：

```
application    指明使用了APPLICATION标签
default:x      设置一个可选整数字段的默认值
explicit       给一个隐式的标签设置一个额外的显式标签
optional       标记字段为ASN.1 OPTIONAL的
set            表示期望一个SET而不是SEQUENCE类型
tag:x          指定ASN.1标签码，隐含ASN.1 CONTEXT SPECIFIC
```

如果结构体的第一个字段的类型为RawContent，则会将原始ASN1结构体内容包存在该字段。

如果切片成员的类型名以"SET"结尾，则视为该字段有"set"标签。这是给不能使用标签的嵌套切片使用的。

其它ASN.1类型不支持，如果遭遇这些类型，Unmarshal返回解析错误。

func UnmarshalWithParams

```
func UnmarshalWithParams(b []byte, val interface{}, params string)
```

UnmarshalWithParams允许指定val顶层成员的字段参数，格式和字段标签相同。

package base32

```
import "encoding/base32"
```

base32包实现了RFC 4648规定的base32编码。

Index

Examples

Variables

```
var HexEncoding = NewEncoding(encodeHex)
```

RFC 4648定义的“扩展Hex字符集”，用于DNS。

```
var StdEncoding = NewEncoding(encodeStd)
```

RFC 4648定义的标准base32编码字符集。

type CorruptInputError

```
type CorruptInputError int64
```

func (CorruptInputError) Error

```
func (e CorruptInputError) Error() string
```

type Encoding

```
type Encoding struct {
    // 内含隐藏或非导出字段
}
```

双向的编码/解码协议，根据一个32字符的字符集定义，RFC 4648标准化了两种字符集。默认字符集用于SASI和GSSAPI，另一种用于DNSSEC。

func NewEncoding

```
func NewEncoding(encoder string) *Encoding
```

使用给出的字符集生成一个*Encoding，字符集必须是32字节的字符串。

func (*Encoding) DecodedLen

```
func (enc *Encoding) DecodedLen(n int) int
```

返回n字节base32编码的数据解码后的最大长度。

func (*Encoding) Decode

```
func (enc *Encoding) Decode(dst, src []byte) (n int, err error)
```

将src的数据解码后存入dst，最多写DecodedLen(len(src))字节数据到dst，并返回写入的字节数。如果src包含非法字符，将返回成功写入的字符数和CorruptInputError。换行符（\r、\n）会被忽略。

func (*Encoding) DecodeString

```
func (enc *Encoding) DecodeString(s string) ([]byte, error)
```

返回base32编码的字符串s代表的数据。

Example

```
str := "ONXW2ZJAMRQXIYJAO5UXI2BAAAQGC3TEEDX3XPY="
data, err := base32.StdEncoding.DecodeString(str)
if err != nil {
    fmt.Println("error:", err)
    return
}
fmt.Printf("%q\n", data)
```

Output:

```
"some data with \x00 and \ufeff"
```

func (*Encoding) EncodedLen

```
func (enc *Encoding) EncodedLen(n int) int
```

返回n字节数据进行base32编码后的最大长度。

func (*Encoding) Encode

```
func (enc *Encoding) Encode(dst, src []byte)
```

将src的数据编码后存入dst，最多写EncodedLen(len(src))字节数据到dst，并返回写入的字节数。

函数会把输出设置为8的倍数，因此不建议对大数据流的独立数据块执行此方法，使用NewEncoder()代替。

func (*Encoding) EncodeToString

```
func (enc *Encoding) EncodeToString(src []byte) string
```

返回将src编码后的字符串。

Example

```
data := []byte("any + old & data")
str := base32.StdEncoding.EncodeToString(data)
fmt.Println(str)
```

Output:

```
MFXHSIBLEBXWYZBAEYQGIYLUME======
```

func NewDecoder

```
func NewDecoder(enc *Encoding, r io.Reader) io.Reader
```

创建一个新的base32流解码器。

func NewEncoder

```
func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser
```

创建一个新的base32流编码器。写入的数据会在编码后再写入w，base32编码每5字节执行一次编码操作；写入完毕后，使用者必须调用Close方法以便将未写入的数据从缓存中刷新到w中。

Example

```
input := []byte("foo\x00bar")
encoder := base32.NewEncoder(base32.StdEncoding, os.Stdout)
encoder.Write(input)
// Must close the encoder when finished to flush any partial blocks
// If you comment out the following line, the last partial block "r
// won't be encoded.
encoder.Close()
```

Output:

```
MZXW6ADCMFZA====
```

package base64

```
import "encoding/base64"
```

base64实现了RFC 4648规定的base64编码。

Index

Examples

Variables

```
var StdEncoding = NewEncoding(encodeStd)
```

RFC 4648定义的标准base64编码字符集。

```
var URLEncoding = NewEncoding(encodeURL)
```

RFC 4648定义的另一base64编码字符集，用于URL和文件名。

type CorruptInputError

```
type CorruptInputError int64
```

func (CorruptInputError) Error

```
func (e CorruptInputError) Error() string
```

type Encoding

```
type Encoding struct {
    // 内含隐藏或非导出字段
}
```

双向的编码/解码协议，根据一个64字符的字符集定义，RFC 4648标准化了两种字符集。默认字符集用于MIME（RFC 2045）和PEM（RFC 1421）编码；另一种用于URL和文件名，用'-'和'_'替换了'+'和'/'。

func NewEncoding

```
func NewEncoding(encoder string) *Encoding
```

使用给出的字符集生成一个*Encoding，字符集必须是64字节的字符串。

func (*Encoding) DecodedLen

```
func (enc *Encoding) DecodedLen(n int) int
```

返回n字节base64编码的数据解码后的最大长度。

func (*Encoding) Decode

```
func (enc *Encoding) Decode(dst, src []byte) (n int, err error)
```

将src的数据解码后存入dst，最多写DecodedLen(len(src))字节数据到dst，并返回写入的字节数。 如果src包含非法字符，将返回成功写入的字符数和CorruptInputError。换行符（\r、\n）会被忽略。

func (*Encoding) DecodeString

```
func (enc *Encoding) DecodeString(s string) ([]byte, error)
```

返回base64编码的字符串s代表的数据。

Example

```
str := "c29tZSBkYXRhIHdpdGggACBhbmQg77u/"
data, err := base64.StdEncoding.DecodeString(str)
if err != nil {
    fmt.Println("error:", err)
    return
}
fmt.Printf("%q\n", data)
```

Output:

```
"some data with \x00 and \ufeff"
```

func (*Encoding) EncodedLen

```
func (enc *Encoding) EncodedLen(n int) int
```

返回n字节数据进行base64编码后的最大长度。

func (*Encoding) Encode

```
func (enc *Encoding) Encode(dst, src []byte)
```

将src的数据编码后存入dst，最多写EncodedLen(len(src))字节数据到dst，并返回写入的字节数。

函数会把输出设置为4的倍数，因此不建议对大数据流的独立数据块执行此方法，使用NewEncoder()代替。

func (*Encoding) EncodeToString

```
func (enc *Encoding) EncodeToString(src []byte) string
```

返回将src编码后的字符串。

Example

```
data := []byte("any + old & data")
str := base64.StdEncoding.EncodeToString(data)
fmt.Println(str)
```

Output:

```
YW55ICsgb2xkICYgZGF0YQ==
```

func NewDecoder

```
func NewDecoder(enc *Encoding, r io.Reader) io.Reader
```

创建一个新的base64流解码器。

func NewEncoder

```
func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser
```

创建一个新的base64流编码器。写入的数据会在编码后再写入w，base32编码每3字节执行一次编码操作；写入完毕后，使用者必须调用Close方法以便将未写入的数据从缓存中刷新到w中。

Example

```
input := []byte("foo\x00bar")
encoder := base64.NewEncoder(base64.StdEncoding, os.Stdout)
encoder.Write(input)
// Must close the encoder when finished to flush any partial blocks
// If you comment out the following line, the last partial block "r
// won't be encoded.
encoder.Close()
```

Output:

```
Zm9vAGJhcg==
```

package binary

```
import "encoding/binary"
```

binary包实现了简单的数字与字节序列的转换以及变长值的编解码。

数字翻译为定长值来读写，一个定长值，要么是固定长度的数字类型（int8, uint8, int16, float32, complex64, ...）或者只包含定长值的结构体或者数组。

变长值是使用一到多个字节编码整数的方法，绝对值较小的数字会占用较少的字节数。详情请参见：http://code.google.com/apis/protocolbuffers/docs/encoding.html。

本包相对于效率更注重简单。如果需要高效的序列化，特别是数据结构较复杂的，请参见更高级的解决方法，例如encoding/gob包，或者采用协议缓存。

Index

Examples

Constants

```
const (
    MaxVarintLen16 = 3
    MaxVarintLen32 = 5
    MaxVarintLen64 = 10
)
```

变长编码N位整数的最大字节数。

Variables

```
var BigEndian bigEndian
```

大端字节序的实现。

```
var LittleEndian littleEndian
```

小端字节序的实现。

type ByteOrder

```
type ByteOrder interface {
    Uint16([]byte) uint16
    Uint32([]byte) uint32
    Uint64([]byte) uint64
    PutUint16([]byte, uint16)
    PutUint32([]byte, uint32)
    PutUint64([]byte, uint64)
    String() string
}
```

ByteOrder规定了如何将字节序列和 16、32或64比特的无符号整数互相转化。

func Size

```
func Size(v interface{}) int
```

返回v编码后会占用多少字节，注意v必须是定长值、定长值的切片、定长值的指针。

func Uvarint

```
func Uvarint(buf []byte) (uint64, int)
```

从buf解码一个uint64，返回该数字和读取的字节长度，如果发生了错误，该数字为0而读取长度n返回值的意思是：

```
    n == 0: buf不完整，太短了
    n  < 0: 值太大了，64比特装不下（溢出），-n为读取的字节数
```

func Varint

```
func Varint(buf []byte) (int64, int)
```

从buf解码一个int64，返回该数字和读取的字节长度，如果发生了错误，该数字为0而读取长度n返回值的意思是：

```
    n == 0: buf不完整，太短了
    n  < 0: 值太大了，64比特装不下（溢出），-n为读取的字节数
```

func ReadUvarint

```
func ReadUvarint(r io.ByteReader) (uint64, error)
```

从r读取一个编码后的无符号整数，并返回该整数。

func ReadVarint

```
func ReadVarint(r io.ByteReader) (int64, error)
```

从r读取一个编码后的有符号整数，并返回该整数。

func PutUvarint

```
func PutUvarint(buf []byte, x uint64) int
```

将一个uint64数字编码写入buf并返回写入的长度，如果buf太小，则会panic。

func PutVarint

```
func PutVarint(buf []byte, x int64) int
```

将一个int64数字编码写入buf并返回写入的长度，如果buf太小，则会panic。

func Read

```
func Read(r io.Reader, order ByteOrder, data interface{}) error
```

从r中读取binary编码的数据并赋给data，data必须是一个指向定长值的指针或者定长值的切片。从r读取的字节使用order指定的字节序解码并写入data的字段里当写入结构体是，名字中有'_'的字段会被跳过，这些字段可用于填充（内存空间）。

Example

```
var pi float64
b := []byte{0x18, 0x2d, 0x44, 0x54, 0xfb, 0x21, 0x09, 0x40}
buf := bytes.NewReader(b)
err := binary.Read(buf, binary.LittleEndian, &pi)
if err != nil {
    fmt.Println("binary.Read failed:", err)
}
fmt.Print(pi)
```

Output:

```
3.141592653589793
```

func Write

```
func Write(w io.Writer, order ByteOrder, data interface{}) error
```

将data的binary编码格式写入w，data必须是定长值、定长值的切片、定长值的指针。order指定写入数据的字节序，写入结构体时，名字中有'_'的字段会置为0。

Example

```
buf := new(bytes.Buffer)
var pi float64 = math.Pi
err := binary.Write(buf, binary.LittleEndian, pi)
if err != nil {
    fmt.Println("binary.Write failed:", err)
}
fmt.Printf("% x", buf.Bytes())
```

Output:

```
18 2d 44 54 fb 21 09 40
```

Example (Multi)

```
buf := new(bytes.Buffer)
var data = []interface{}{
    uint16(61374),
    int8(-54),
    uint8(254),
}
for _, v := range data {
    err := binary.Write(buf, binary.LittleEndian, v)
    if err != nil {
        fmt.Println("binary.Write failed:", err)
    }
}
fmt.Printf("%x", buf.Bytes())
```

Output:

```
beefcafe
```

package csv

`import "encoding/csv"`

csv读写逗号分隔值（csv）的文件。

一个csv分拣包含零到多条记录，每条记录一到多个字段。每个记录用换行符分隔。最后一条记录后面可以有换行符，也可以没有。

```
field1,field2,field3
```

空白视为字段的一部分。

换行符前面的回车符会悄悄的被删掉。

忽略空行。只有空白的行（除了末尾换行符之外）不视为空行。

以双引号"开始和结束的字段成为受引字段，其开始和结束的引号不属于字段。

资源：

```
normal string,"quoted-field"
```

产生如下字段：

```
{`normal string`, `quoted-field`}
```

受引字段内部，如果有两个连续的双引号，则视为一个单纯的双引号字符：

```
"the ""word"" is true","a ""quoted-field"""
```

生成：

```
{`the "word" is true`, `a "quoted-field"`}
```

受引字段里可以包含换行和逗号：

```
"Multi-line
field","comma is ,"
```

生成：

```
{`Multi-line
field`, `comma is ,`}
```

Index

Variables

```
var (
    ErrTrailingComma = errors.New("extra delimiter at end of line")
    ErrBareQuote     = errors.New("bare \" in non-quoted-field")
    ErrQuote         = errors.New("extraneous \" in field")
    ErrFieldCount    = errors.New("wrong number of fields in line")
)
```

这些都是PaserError.Err字段可能的值。

type ParseError

```
type ParseError struct {
    Line   int   // 出错的行号
    Column int   // 出错的列号
    Err    error // 具体的错误
}
```

当解析错误时返回ParseError，第一个行为1，第一列为0。

func (*ParseError) Error

```
func (e *ParseError) Error() string
```

type Reader

```
type Reader struct {
    Comma            rune // 字段分隔符（NewReader将之设为','）
    Comment          rune // 一行开始位置的注释标识符
    FieldsPerRecord  int  // 每条记录期望的字段数
    LazyQuotes       bool // 允许懒引号
    TrailingComma    bool // 忽略，出于后端兼容性而保留
    TrimLeadingSpace bool // 去除前导的空白
    // 内含隐藏或非导出字段
}
```

Reader从csv编码的文件中读取记录。

NewReader返回的*Reader期望输入符合RFC 4180。在首次调用Read或者ReadAll之前可以设定导出字段的细节。

Comma是字段分隔符，默认为','。Comment如果不是0，则表示注释标识符，以Comment开始的行会被忽略。如果FieldsPerRecord大于0，Read方法要求每条记录都有给定数目的字段。如果FieldsPerRecord等于0，Read方法会将其设为第一条记录的字段数，因此其余的记录必须有同样数目的字段。如果FieldsPerRecord小于0，不会检查字段数，允许记录有不同数量的字段。如果LazyQuotes为真，引号可以出现在非受引字段里，不连续两个的引号可以出现在受引字段里。如果TrimLeadingSpace为真，字段前导的空白会忽略掉。

func NewReader

```
func NewReader(r io.Reader) *Reader
```

NewReader函数返回一个从r读取的*Reader。

func (*Reader) Read

```
func (r *Reader) Read() (record []string, err error)
```

Read从r读取一条记录，返回值record是字符串的切片，每个字符串代表一个字段。

func (*Reader) ReadAll

```
func (r *Reader) ReadAll() (records [][]string, err error)
```

ReadAll从r中读取所有剩余的记录，每个记录都是字段的切片，成功的调用返回值err为nil而不是EOF。因为ReadAll方法定义为读取直到文件结尾，因此它不会将文件结尾视为应该报告的错误。

type Writer

```
type Writer struct {
    Comma   rune // 字段分隔符（NewWriter将之设为','）
    UseCRLF bool // 如为真，则换行时使用\r\n
    // 内含隐藏或非导出字段
}
```

Writer类型的值将记录写入一个csv编码的文件。

NewWriter返回的*Writer写入记录时，以换行结束记录，用','分隔字段。在第一次调用Write或WriteAll之前，可以设置导出字段的细节。

Comma是字段分隔符。如果UseCRLF为真，Writer在每条记录结束时用\r\n代替\n。

func NewWriter

```
func NewWriter(w io.Writer) *Writer
```

NewWriter返回一个写入w的*Writer。

func (*Writer) Write

```
func (w *Writer) Write(record []string) (err error)
```

向w中写入一条记录，会自行添加必需的引号。记录是字符串切片，每个字符串代表一个字段。

func (*Writer) WriteAll

```
func (w *Writer) WriteAll(records [][]string) (err error)
```

WriteAll方法使用Write方法向w写入多条记录，并在最后调用Flush方法清空缓存。

func (*Writer) Flush

```
func (w *Writer) Flush()
```

将缓存中的数据写入底层的io.Writer。要检查Flush时是否发生错误的话，应调用Error方法。

func (*Writer) Error

```
func (w *Writer) Error() error
```

Error返回在之前的Write方法和Flush方法执行时出现的任何错误。

package gob

```
import "encoding/gob"
```

gob包管理gob流——在编码器（发送器）和解码器（接受器）之间交换的binary值。一般用于传递远端程序调用（RPC）的参数和结果，如net/rpc包就有提供。

本实现给每一个数据类型都编译生成一个编解码程序，当单个编码器用于传递数据流时，会分期偿还编译的消耗，是效率最高的。

Basics

Gob流是自解码的。流中的所有数据都有前缀（采用一个预定义类型的集合）指明其类型。指针不会传递，而是传递值；也就是说数据是压平了的。递归的类型可以很好的工作，但是递归的值（比如说值内某个成员直接/间接指向该值）会出问题。这个问题将来可能会修复。

要使用gob，先要创建一个编码器，并向其一共一系列数据：可以是值，也可以是指向实际存在数据的指针。编码器会确保所有必要的类型信息都被发送。在接收端，解码器从编码数据流中恢复数据并将它们填写进本地变量里。

Types and Values

发送端和接收端的值/类型不需要严格匹配。对结构体来说，字段（根据字段名识别）如果发送端有而接收端没有，会被忽略；接收端有而发送端没有的字段也会被忽略；发送端和接收端都有的字段其类型必须是可兼容的；发送端和接收端都会在gob流和实际go类型之间进行必要的指针取址/寻址工作。举例如下：

```
struct { A, B int }
```

可以和如下类型互相发送和接收：

```
struct { A, B int }    // 同一类型
*struct { A, B int }    // 结构体需要额外重定向（指针）
struct { *A, **B int }    // 字段需要额外重定向（指针）
struct { A, B int64 }    // 同为整型/浮点型且符号类型相同的不同值类型，参见
```

可以发送给如下任一类型：

```
struct { A, B int }    // 同一类型
struct { B, A int }    // 字段顺序改变无影响，按名称匹配
struct { A, B, C int }    // 忽略多出的字段C
struct { B int }    // 忽略缺少的字段A，会丢弃A的值
struct { B, C int }    // 忽略缺少的字段A，忽略多出的字段C
```

但尝试发送给如下类型的话就会导致错误：

```
struct { A int; B uint }    // B字段改变了符号类型
struct { A int; B float }    // B字段改变了类型
struct { }            // 无共同字段名
struct { C, D int }        // 无共同字段名
```

整数以两种方式传递：任意精度有符号整数和任意精度无符号整数。Gob里只有无符号和有符号整数的区别，没有int8、int16等类型的区分。如下所述，发送端会以变长编码发送整数值；接收端接收整数并保存在目标变量里。浮点数则总是使用IEEE-754 64位精度（参见下述）。

有符号整数可以被任意有符号整形接收：int、int16等；无符号整数可以被任意无符号整形接收；浮点数可以被任意浮点数类型接收。但是，接收端类型必须能容纳该值（上溢/下溢都不行），否则解码操作会失败。

结构体、数组和切片都被支持。结构体只编码和解码导出字段。字符串和byte数组/切片有专门的高效表示（参见下述）。当解码切片时，如果当前切片的容量足够会被复用，否则会申请新的底层数组（所以还是用切片地址为好）。此外，生成的切片的长度会修改为解码的成员的个数。

Gob流不支持函数和通道。试图在最顶层编码这些类型的值会导致失败。结构体中包含函数或者通道类型的字段的话，会视作非导出字段（忽略）处理。

Gob可以编码任意实现了GobEncoder接口或者encoding.BinaryMarshaler接口的类型的值（通过调用对应的方法），GobEncoder接口优先。

Gob可以解码任意实现了GobDecoder接口或者encoding.BinaryUnmarshaler接口的类型的值（通过调用对应的方法），同样GobDecoder接口优先。

Encoding Details

这部分文档是编码细节，对多数使用者并不重要。细节是按从底向上的顺序展示的。

无符号整数用两种方法发送。如果该整数小于128，则以一个字节发送该值；否则采用最小长度大端在前的字节流保存该整数，并在最前面使用一个字节保存字节流的字节数相反数。因此0被发送为(00)，7被发送为(07)，而256被发送为(FE 01 00)（字节数2，其相反数-2，用补码表示，为FE）。

布尔值按无符号整数编码，0表示假，1表示真。

有符号整数翻译为一个无符号整数（i=>u）来编码。u的最低字位表示值的符号（以及是否应对值按位取反）；其余位表示值。编码算法表示为如下（非实际代码）：

```
uint u;
if i < 0 {
    u = (^i << 1) | 1    // i按位取反，左移1位，第1位设为1
} else {
    u = (i << 1)    // i不进行取反，左移1位，第1位为0
}
encodeUnsigned(u)
```

这样一来，最低位就相当于标志位，但还会对负数按位取反，以便保证负数不会出现特殊情况（补码表示的负数，其表示会受到整数类型的影响）。如，-129=^128=(^256>>1)编码为(FE 01 01)。

浮点数的数值，首先总是转换float64类型值，该值使用math.Float64bits 函数转换为一个uint64整数，然后将字节序颠倒过来，最后作为一个普通无符号数编码传输。颠倒字节序说明数字的指数和高精度位数部分首先传送。因为低位一般是0，这样可以节约编码的字节数。例如，17.0编码后只有三个字节(FE 31 40)。

字符串和byte数组/切片发送为一个无符号整数指定的字节数后跟不作处理的字节序列。

其它切片和数组都发送为一个无符号整数指定的成员个数后跟所有成员的递归表示的gob编码。

映射发送为一个无符号整数指定的键值对数后给许多键和值的gob编码。非nil但无成员的映射也会发送，因此如果发送者申请了一个映射，接收方也会申请一个映射，即使该映射内没有元素。

结构体也以键值对（字段名：字段值）序列的形式发送。字段值采用递归表示的gob编码发送。如果字段为其类型的零值，则该字段不会被发送。字段编号由编码的结构体的类型确定：编码类型的第一个字段为字段0，第二个为字段1，依次类推。当编码一个结构体的值时，字段编号出于效率考虑是增量编码的，字段总是按字段编号递增的顺序被编码，增量是无符号整数。增量编码将字段编号初始化为-1，因此无符号整型值为7的字段0编码为增量1值7。最后，所有的字段都被发送后，会发送终止标记表明结构体的结束。终止标记为一个增量为0的值，其表示为(00)。

接口类型不会检查兼容性；所有的接口都被视为同一种“接口”类型来传输；类似整数和字节切片，它们都被视为interface{}类型。接口值发送为一个表示其具体类型的字符串标志符（必须由调用者预先注册的名称）后跟表示后续数据字节数的无符号整数（以便需要时可以跳过该值），再后跟保存在接口里的值的动态具体类型的gob编码。nil接口值会发送为标志符为空字符串、不发送值的接口。在接收到之后，由解码器验证该值是否满足接收端变量接口。

类型的表示如下所示。当一个编码器和解码器的连接中定义了一个类型时，该类型会被指定一个整数类型ID。当调用Encoder.Encode(v)时，该方法会确保v及v所有成员都有对应ID，然后本方法会发送一系列对（typeid，encoded-v），其中typeid

是编码类型的类型ID，encoded-v是值v的gob编码。

为了定义一个类型，编码器会选择一个未使用的正数作为id并发送对（-type id, encoded-type），其中encoded-type 是由如下类型构成、描述该类型的wireType类型的gob编码。

```
type wireType struct {
    ArrayT  *ArrayType
    SliceT  *SliceType
    StructT *StructType
    MapT    *MapType
}
type arrayType struct {
    CommonType
    Elem typeId
    Len  int
}
type CommonType struct {
    Name string // 结构体的名字
    Id  int    // 类型的ID
}
type sliceType struct {
    CommonType
    Elem typeId
}
type structType struct {
    CommonType
    Field []*fieldType // 结构体的字段
}
type fieldType struct {
    Name string // 字段名
    Id   int    // 字段的类型ID，必须是已经定义的类型
}
type mapType struct {
    CommonType
    Key  typeId
    Elem typeId
}
```

如果有嵌套的类型，必须先定义所有内部类型的ID才能定义顶层类型的ID用于描述encoded-v。

为了简化启动，有些类型是预先定义好了的，这些类型及其ID如下：

```
bool        1
int         2
uint        3
float       4
[]byte      5
string      6
complex     7
interface   8
// 空缺用于保留ID
WireType    16
ArrayType   17
CommonType  18
SliceType   19
StructType  20
FieldType   21
// 22是[]fieldType类型的ID
MapType     23
```

最后，每一次调用Encode创建的信息都会以一个编码了的无符号整数指明消息剩余部分的字节数。在开始的类型名后面，接口值也以同样的方式包装；事实上，接口值表现的就像对Encode进行递归调用一样。

总的来说，一个gob流看起来是这样的：

```
(byteCount (-type id, encoding of a wireType)* (type id, encoding o
```

其中*表示0或多次重复，值的类型id必须是预定义的，或者在流中值的前面定义。

参见"Gobs of data"获取gob wire格式的设计讨论：

http://golang.org/doc/articles/gobs_of_data.html

Example (Basic)

```
package gob_test
import (
    "bytes"
    "encoding/gob"
    "fmt"
    "log"
)
type P struct {
    X, Y, Z int
    Name    string
}
type Q struct {
    X, Y *int32
    Name string
}
// This example shows the basic usage of the package: Create an enc
// transmit some values, receive them with a decoder.
func Example_basic() {
    // Initialize the encoder and decoder.  Normally enc and dec wo
    // bound to network connections and the encoder and decoder wou
    // run in different processes.
    var network bytes.Buffer        // Stand-in for a network conne
    enc := gob.NewEncoder(&network) // Will write to network.
    dec := gob.NewDecoder(&network) // Will read from network.
    // Encode (send) some values.
    err := enc.Encode(P{3, 4, 5, "Pythagoras"})
    if err != nil {
        log.Fatal("encode error:", err)
    }
    err = enc.Encode(P{1782, 1841, 1922, "Treehouse"})
    if err != nil {
        log.Fatal("encode error:", err)
    }
    // Decode (receive) and print the values.
    var q Q
    err = dec.Decode(&q)
    if err != nil {
        log.Fatal("decode error 1:", err)
    }
    fmt.Printf("%q: {%d, %d}\n", q.Name, *q.X, *q.Y)
    err = dec.Decode(&q)
    if err != nil {
        log.Fatal("decode error 2:", err)
    }
    fmt.Printf("%q: {%d, %d}\n", q.Name, *q.X, *q.Y)
    // Output:
    // "Pythagoras": {3, 4}
    // "Treehouse": {1782, 1841}
}
```

Example (EncodeDecode)

```
package gob_test
import (
    "bytes"
    "encoding/gob"
    "fmt"
    "log"
)
// The Vector type has unexported fields, which the package cannot 
// We therefore write a BinaryMarshal/BinaryUnmarshal method pair t
// to send and receive the type with the gob package. These interfa
// defined in the "encoding" package.
// We could equivalently use the locally defined GobEncode/GobDecod
// interfaces.
type Vector struct {
    x, y, z int
}
func (v Vector) MarshalBinary() ([]byte, error) {
    // A simple encoding: plain text.
    var b bytes.Buffer
    fmt.Fprintln(&b, v.x, v.y, v.z)
    return b.Bytes(), nil
}
// UnmarshalBinary modifies the receiver so it must take a pointer 
func (v *Vector) UnmarshalBinary(data []byte) error {
    // A simple encoding: plain text.
    b := bytes.NewBuffer(data)
    _, err := fmt.Fscanln(b, &v.x, &v.y, &v.z)
    return err
}
// This example transmits a value that implements the custom encodi
func Example_encodeDecode() {
    var network bytes.Buffer // Stand-in for the network.
    // Create an encoder and send a value.
    enc := gob.NewEncoder(&network)
    err := enc.Encode(Vector{3, 4, 5})
    if err != nil {
        log.Fatal("encode:", err)
    }
    // Create a decoder and receive a value.
    dec := gob.NewDecoder(&network)
    var v Vector
    err = dec.Decode(&v)
    if err != nil {
        log.Fatal("decode:", err)
    }
    fmt.Println(v)
    // Output:
    // {3 4 5}
}
```

Example (Interface)

```
package gob_test
import (
    "bytes"
    "encoding/gob"
    "fmt"
    "log"
    "math"
)
type Point struct {
    X, Y int
}
func (p Point) Hypotenuse() float64 {
    return math.Hypot(float64(p.X), float64(p.Y))
}
type Pythagoras interface {
    Hypotenuse() float64
}
// This example shows how to encode an interface value. The key
// distinction from regular types is to register the concrete type
// implements the interface.
func Example_interface() {
    var network bytes.Buffer // Stand-in for the network.
    // We must register the concrete type for the encoder and decod
    // normally be on a separate machine from the encoder). On each
    // engine which concrete type is being sent that implements the
    gob.Register(Point{})
    // Create an encoder and send some values.
    enc := gob.NewEncoder(&network)
    for i := 1; i <= 3; i++ {
        interfaceEncode(enc, Point{3 * i, 4 * i})
    }
    // Create a decoder and receive some values.
    dec := gob.NewDecoder(&network)
    for i := 1; i <= 3; i++ {
        result := interfaceDecode(dec)
        fmt.Println(result.Hypotenuse())
    }
    // Output:
    // 5
    // 10
    // 15
}
// interfaceEncode encodes the interface value into the encoder.
func interfaceEncode(enc *gob.Encoder, p Pythagoras) {
    // The encode will fail unless the concrete type has been
    // registered. We registered it in the calling function.
    // Pass pointer to interface so Encode sees (and hence sends) a
    // interface type.  If we passed p directly it would see the co
    // See the blog post, "The Laws of Reflection" for background.
    err := enc.Encode(&p)
```

```
    if err != nil {
        log.Fatal("encode:", err)
    }
}
// interfaceDecode decodes the next interface value from the strear
func interfaceDecode(dec *gob.Decoder) Pythagoras {
    // The decode will fail unless the concrete type on the wire ha
    // registered. We registered it in the calling function.
    var p Pythagoras
    err := dec.Decode(&p)
    if err != nil {
        log.Fatal("decode:", err)
    }
    return p
}
```

Index

Examples

type GobDecoder

```
type GobDecoder interface {
    // GobDecode必须是指针的方法，使用切片数据重写指针指向的值
    // 切片数据应该由同一具体类型的GobEncode生成
    GobDecode([]byte) error
}
```

GobDecoder是一个描述数据的接口，提供自己的方案来解码GobEncoder发送的数据。

type GobEncoder

```
type GobEncoder interface {
    // GobEncode方法返回一个代表值的切片数据
    // 该切片数据由同一类型的指针方法GobDecoder接受解码
    GobEncode() ([]byte, error)
}
```

GobEncoder是一个描述数据的接口，提供自己的方案来将数据编码供GobDecoder接收并解码。一个实现了GobEncoder接口和GobDecoder接口的类型可以完全控制自身数据的表示，因此可以包含非导出字段、通道、函数等数据，这些数据gob流正常是不能传输的。

注意：因为gob数据可以被永久保存，在软件更新的过程中保证用于GobEncoder编码的数据的稳定性（保证兼容）是较好的设计原则。例如，让GobEncode 接口在编码后数据里包含版本信息可能很有意义。

func Register

```
func Register(value interface{})
```

Register记录value下层具体值的类型和其名称。该名称将用来识别发送或接受接口类型值时下层的具体类型。本函数只应在初始化时调用，如果类型和名字的映射不是一一对应的，会panic。

func RegisterName

```
func RegisterName(name string, value interface{})
```

RegisterName类似Register，淡灰使用提供的name代替类型的默认名称。

type Decoder

```
type Decoder struct {
    // 内含隐藏或不导出字段
}
```

Decoder管理从连接远端读取的类型和数据信息的解释（的操作）。

func NewDecoder

```
func NewDecoder(r io.Reader) *Decoder
```

函数返回一个从r读取数据的*Decoder，如果r不满足io.ByteReader接口，则会包装r为bufio.Reader。

func (*Decoder) Decode

```
func (dec *Decoder) Decode(e interface{}) error
```

Decode从输入流读取下一个之并将该值存入e。如果e是nil，将丢弃该值；否则e必须是可接收该值的类型的指针。如果输入结束，方法会返回io.EOF并且不修改e（指向的值）。

func (*Decoder) DecodeValue

```
func (dec *Decoder) DecodeValue(v reflect.Value) error
```

DecodeValue从输入流读取下一个值，如果v是reflect.Value类型的零值（v.Kind() == Invalid），方法丢弃该值；否则它会把该值存入v。此时，v必须代表一个非nil的指向实际存在值的指针或者可写入的reflect.Value（v.CanSet()为真）。如果输入结束，方法会返回io.EOF并且不修改e（指向的值）。

type Encoder

```
type Encoder struct {
    // 内含隐藏或不导出字段
}
```

Encoder管理将数据和类型信息编码后发送到连接远端（的操作）。

func NewEncoder

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder返回一个将编码后数据写入w的*Encoder。

func (*Encoder) Encode

```
func (enc *Encoder) Encode(e interface{}) error
```

Encode方法将e编码后发送，并且会保证所有的类型信息都先发送。

func (*Encoder) EncodeValue

```
func (enc *Encoder) EncodeValue(value reflect.Value) error
```

EncodeValue方法将value代表的数据编码后发送，并且会保证所有的类型信息都先发送。

type CommonType

```
type CommonType struct {
    Name string
    Id   typeId
}
```

CommonType保存所有类型的成员。这是个历史遗留“问题”，出于保持binary兼容性才保留，只用于类型描述的编码。该类型应视为不导出类型。

package hex

```
import "encoding/hex"
```

hex包实现了16进制字符表示的编解码。

Index

Variables

```
var ErrLength = errors.New("encoding/hex: odd length hex string")
```

解码一个长度为奇数的切片时，将返回此错误。

type InvalidByteError

```
type InvalidByteError byte
```

描述一个hex编码字符串中的非法字符。

func (InvalidByteError) Error

```
func (e InvalidByteError) Error() string
```

func DecodedLen

```
func DecodedLen(x int) int
```

长度x的编码数据解码后的明文数据的长度

func Decode

```
func Decode(dst, src []byte) (int, error)
```

将src解码为DecodedLen(len(src))字节，返回实际写入dst的字节数；如遇到非法字符，返回描述错误的error。

func DecodeString

```
func DecodeString(s string) ([]byte, error)
```

返回hex编码的字符串s代表的数据。

func EncodedLen

```
func EncodedLen(n int) int
```

长度x的明文数据编码后的编码数据的长度。

func Encode

```
func Encode(dst, src []byte) int
```

将src的数据解码为EncodedLen(len(src))字节，返回实际写入dst的字节数：EncodedLen(len(src))。

func EncodeToString

```
func EncodeToString(src []byte) string
```

将数据src编码为字符串s。

func Dump

```
func Dump(data []byte) string
```

返回给定数据的hex dump格式的字符串，这个字符串与控制台下`hexdump -C`对该数据的输出是一致的。

func Dumper

```
func Dumper(w io.Writer) io.WriteCloser
```

返回一个io.WriteCloser接口，将写入的数据的hex dump格式写入w，具体格式为'hexdump -C'。

package json

```
import "encoding/json"
```

json包实现了json对象的编解码，参见RFC 4627。Json对象和go类型的映射关系请参见Marshal和Unmarshal函数的文档。

参见"JSON and Go"获取本包的一个介绍：http://golang.org/doc/articles/json_and_go.html

Index

Examples

type InvalidUTF8Error

```
type InvalidUTF8Error struct {
    S string // 引发错误的完整字符串
}
```

Go 1.2之前版本，当试图编码一个包含非法utf-8序列的字符串时会返回本错误。Go 1.2及之后版本，编码器会强行将非法字节替换为unicode字符U+FFFD来使字符串合法。本错误已不会再出现，但出于向后兼容考虑而保留。

func (*InvalidUTF8Error) Error

```
func (e *InvalidUTF8Error) Error() string
```

type InvalidUnmarshalError

```
type InvalidUnmarshalError struct {
    Type reflect.Type
}
```

InvalidUnmarshalError用于描述一个传递给解码器的非法参数。（解码器的参数必须是非nil指针）

func (*InvalidUnmarshalError) Error

```
func (e *InvalidUnmarshalError) Error() string
```

type SyntaxError

```
type SyntaxError struct {
    Offset int64 // 在读取Offset个字节后出现错误
    // 内含隐藏或非导出字段
}
```

SyntaxError表示一个json语法错误。

func (*SyntaxError) Error

```
func (e *SyntaxError) Error() string
```

type UnmarshalFieldError

```
type UnmarshalFieldError struct {
    Key   string
    Type  reflect.Type
    Field reflect.StructField
}
```

UnmarshalFieldError表示一个json对象的键指向一个非导出字段。（因此不能写入；已不再使用，出于兼容保留）

func (*UnmarshalFieldError) Error

```
func (e *UnmarshalFieldError) Error() string
```

type UnmarshalTypeError

```
type UnmarshalTypeError struct {
    Value string       // 描述json值："bool", "array", "number -5"
    Type  reflect.Type // 不能转化为的go类型
}
```

UnmarshalTypeError表示一个json值不能转化为特定的go类型的值。

func (*UnmarshalTypeError) Error

```
func (e *UnmarshalTypeError) Error() string
```

type UnsupportedTypeError

```
type UnsupportedTypeError struct {
    Type reflect.Type
}
```

UnsupportedTypeError表示试图编码一个不支持类型的值。

func (*UnsupportedTypeError) Error

```
func (e *UnsupportedTypeError) Error() string
```

type UnsupportedValueError

```
type UnsupportedValueError struct {
    Value reflect.Value
    Str   string
}
```

func (*UnsupportedValueError) Error

```
func (e *UnsupportedValueError) Error() string
```

type MarshalerError

```
type MarshalerError struct {
    Type reflect.Type
    Err  error
}
```

func (*MarshalerError) Error

```
func (e *MarshalerError) Error() string
```

type Number

```
type Number string
```

Number类型代表一个json数字字面量。

func (Number) Int64

```
func (n Number) Int64() (int64, error)
```

将该数字作为int64类型返回。

func (Number) Float64

```
func (n Number) Float64() (float64, error)
```

将该数字作为float64类型返回。

func (Number) String

```
func (n Number) String() string
```

返回该数字的字面值文本表示。

type RawMessage

```
type RawMessage []byte
```

RawMessage类型是一个保持原本编码的json对象。本类型实现了Marshaler和Unmarshaler接口，用于延迟json的解码或者预计算json的编码。

Example

```
type Color struct {
    Space string
    Point json.RawMessage // delay parsing until we know the color
}
type RGB struct {
    R   uint8
    G   uint8
    B   uint8
}
type YCbCr struct {
    Y   uint8
    Cb  int8
    Cr  int8
}
var j = []byte(`[
        {"Space": "YCbCr", "Point": {"Y": 255, "Cb": 0, "Cr": -10}}
        {"Space": "RGB",   "Point": {"R": 98, "G": 218, "B": 255}}
    ]`)
var colors []Color
err := json.Unmarshal(j, &colors)
if err != nil {
    log.Fatalln("error:", err)
}
for _, c := range colors {
    var dst interface{}
    switch c.Space {
    case "RGB":
        dst = new(RGB)
    case "YCbCr":
        dst = new(YCbCr)
    }
    err := json.Unmarshal(c.Point, dst)
    if err != nil {
        log.Fatalln("error:", err)
    }
    fmt.Println(c.Space, dst)
}
```

Output:

```
YCbCr &{255 0 -10}
RGB &{98 218 255}
```

func (*RawMessage) MarshalJSON

```
func (m *RawMessage) MarshalJSON() ([]byte, error)
```

MarshalJSON返回*m的json编码。

func (*RawMessage) UnmarshalJSON

```
func (m *RawMessage) UnmarshalJSON(data []byte) error
```

UnmarshalJSON将*m设为data的一个拷贝。

type Marshaler

```
type Marshaler interface {
    MarshalJSON() ([]byte, error)
}
```

实现了Marshaler接口的类型可以将自身序列化为合法的json描述。

type Unmarshaler

```
type Unmarshaler interface {
    UnmarshalJSON([]byte) error
}
```

实现了Unmarshaler接口的对象可以将自身的json描述反序列化。该方法可以认为输入是合法的json字符串。如果要在方法返回后保存自身的json数据，必须进行拷贝。

func Compact

```
func Compact(dst *bytes.Buffer, src []byte) error
```

Compact函数将json编码的src中无用的空白字符剔除后写入dst。

func HTMLEscape

```
func HTMLEscape(dst *bytes.Buffer, src []byte)
```

HTMLEscape 函数将json编码的src中的<、>、&、U+2028 和U+2029字符替换为\u003c、\u003e、\u0026、\u2028、\u2029 转义字符串，以便json编码可以安全的嵌入HTML的<script>标签里。因为历史原因，网络浏览器不支持在<script>标签中使用标准HTML转义， 因此必须使用另一种json编码方案。

func Indent

```
func Indent(dst *bytes.Buffer, src []byte, prefix, indent string) e
```

Indent函数将json编码的调整缩进之后写入dst。每一个json元素/数组都另起一行开始，以prefix为起始，一或多个indent缩进（数目看嵌套层数）。写入dst的数据起始没有prefix字符，也没有indent字符，最后也不换行，因此可以更好的嵌入其他格式化后的json数据里。

Example

```
type Road struct {
    Name   string
    Number int
}
roads := []Road{
    {"Diamond Fork", 29},
    {"Sheep Creek", 51},
}
b, err := json.Marshal(roads)
if err != nil {
    log.Fatal(err)
}
var out bytes.Buffer
json.Indent(&out, b, "=", "\t")
out.WriteTo(os.Stdout)
```

Output:

```
[
=    {
=        "Name": "Diamond Fork",
=        "Number": 29
=    },
=    {
=        "Name": "Sheep Creek",
=        "Number": 51
=    }
=]
```

func Marshal

```
func Marshal(v interface{}) ([]byte, error)
```

Marshal函数返回v的json编码。

Marshal函数会递归的处理值。如果一个值实现了Marshaler接口切非nil指针，会调用其MarshalJSON方法来生成json编码。nil指针异常并不是严格必需的，但会模拟与UnmarshalJSON的行为类似的必需的异常。

否则，Marshal函数使用下面的基于类型的默认编码格式：

布尔类型编码为json布尔类型。

浮点数、整数和Number类型的值编码为json数字类型。

字符串编码为json字符串。角括号"<"和">"会转义为"\u003c"和"\u003e"以避免某些浏览器吧json输出错误理解为HTML。基于同样的原因，"&"转义为"\u0026"。

数组和切片类型的值编码为json数组，但[]byte编码为base64编码字符串，nil切片编码为null。

结构体的值编码为json对象。每一个导出字段变成该对象的一个成员，除非：

```
- 字段的标签是"-"
- 字段是空值，而其标签指定了omitempty选项
```

空值是false、0、""、nil指针、nil接口、长度为0的数组、切片、映射。对象默认键字符串是结构体的字段名，但可以在结构体字段的标签里指定。结构体标签值里的"json"键为键名，后跟可选的逗号和选项，举例如下：

```
// 字段被本包忽略
Field int `json:"-"`
// 字段在json里的键为"myName"
Field int `json:"myName"`
// 字段在json里的键为"myName"且如果字段为空值将在对象中省略掉
Field int `json:"myName,omitempty"`
// 字段在json里的键为"Field"（默认值），但如果字段为空值会跳过；注意前导的逗号
Field int `json:",omitempty"`
```

"string"选项标记一个字段在编码json时应编码为字符串。它只适用于字符串、浮点数、整数类型的字段。这个额外水平的编码选项有时候会用于和javascript程序交互：

```
Int64String int64 `json:",string"`
```

如果键名是只含有unicode字符、数字、美元符号、百分号、连字符、下划线和斜杠的非空字符串，将使用它代替字段名。

匿名的结构体字段一般序列化为他们内部的导出字段就好像位于外层结构体中一样。如果一个匿名结构体字段的标签给其提供了键名，则会使用键名代替字段名，而不视为匿名。

Go结构体字段的可视性规则用于供json决定那个字段应该序列化或反序列化时是经过修正了的。如果同一层次有多个（匿名）字段且该层次是最小嵌套的（嵌套层次则使用默认go规则），会应用如下额外规则：

1）json标签为"-"的匿名字段强行忽略，不作考虑；

2）json标签提供了键名的匿名字段，视为非匿名字段；

3）其余字段中如果只有一个匿名字段，则使用该字段；

4）其余字段中如果有多个匿名字段，但压平后不会出现冲突，所有匿名字段压平；

5）其余字段中如果有多个匿名字段，但压平后出现冲突，全部忽略，不产生错误。

对匿名结构体字段的管理是从go1.1开始的，在之前的版本，匿名字段会直接忽略掉。

映射类型的值编码为json对象。映射的键必须是字符串，对象的键直接使用映射的键。

指针类型的值编码为其指向的值（的json编码）。nil指针编码为null。

接口类型的值编码为接口内保持的具体类型的值（的json编码）。nil接口编码为null。

通道、复数、函数类型的值不能编码进json。尝试编码它们会导致Marshal函数返回UnsupportedTypeError。

Json不能表示循环的数据结构，将一个循环的结构提供给Marshal函数会导致无休止的循环。

Example

```
type ColorGroup struct {
    ID     int
    Name   string
    Colors []string
}
group := ColorGroup{
    ID:     1,
    Name:   "Reds",
    Colors: []string{"Crimson", "Red", "Ruby", "Maroon"},
}
b, err := json.Marshal(group)
if err != nil {
    fmt.Println("error:", err)
}
os.Stdout.Write(b)
```

Output:

```
{"ID":1,"Name":"Reds","Colors":["Crimson","Red","Ruby","Maroon"]}
```

func MarshalIndent

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, e
```

MarshalIndent类似Marshal但会使用缩进将输出格式化。

func Unmarshal

```
func Unmarshal(data []byte, v interface{}) error
```

Unmarshal函数解析json编码的数据并将结果存入v指向的值。

Unmarshal和Marshal做相反的操作，必要时申请映射、切片或指针，有如下的附加规则：

要将json数据解码写入一个指针，Unmarshal函数首先处理json数据是json字面值null的情况。此时，函数将指针设为nil；否则，函数将json数据解码写入指针指向的值；如果指针本身是nil，函数会先申请一个值并使指针指向它。

要将json数据解码写入一个结构体，函数会匹配输入对象的键和Marshal使用的键（结构体字段名或者它的标签指定的键名），优先选择精确的匹配，但也接受大小写不敏感的匹配。

要将json数据解码写入一个接口类型值，函数会将数据解码为如下类型写入接口：

```
Bool                    对应JSON布尔类型
float64                 对应JSON数字类型
string                  对应JSON字符串类型
[]interface{}           对应JSON数组
map[string]interface{}  对应JSON对象
nil                     对应JSON的null
```

如果一个JSON值不匹配给出的目标类型，或者如果一个json数字写入目标类型时溢出，Unmarshal函数会跳过该字段并尽量完成其余的解码操作。如果没有出现更加严重的错误，本函数会返回一个描述第一个此类错误的详细信息的UnmarshalTypeError。

JSON的null值解码为go的接口、指针、切片时会将它们设为nil，因为null在json里一般表示“不存在”。 解码json的null值到其他go类型时，不会造成任何改变，也不会产生错误。

当解码字符串时，不合法的utf-8或utf-16代理（字符）对不视为错误，而是将非法字符替换为unicode字符U+FFFD。

Example

```
var jsonBlob = []byte(`[
        {"Name": "Platypus", "Order": "Monotremata"},
        {"Name": "Quoll",    "Order": "Dasyuromorphia"}
    ]`)
type Animal struct {
    Name  string
    Order string
}
var animals []Animal
err := json.Unmarshal(jsonBlob, &animals)
if err != nil {
    fmt.Println("error:", err)
}
fmt.Printf("%+v", animals)
```

Output:

```
[{Name:Platypus Order:Monotremata} {Name:Quoll Order:Dasyuromorphia
```

type Decoder

```
type Decoder struct {
    // 内含隐藏或非导出字段
}
```

Decoder从输入流解码json对象

Example

```
const jsonStream = `
        {"Name": "Ed", "Text": "Knock knock."}
        {"Name": "Sam", "Text": "Who's there?"}
        {"Name": "Ed", "Text": "Go fmt."}
        {"Name": "Sam", "Text": "Go fmt who?"}
        {"Name": "Ed", "Text": "Go fmt yourself!"}
    `
type Message struct {
    Name, Text string
}
dec := json.NewDecoder(strings.NewReader(jsonStream))
for {
    var m Message
    if err := dec.Decode(&m); err == io.EOF {
        break
    } else if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%s: %s\n", m.Name, m.Text)
}
```

Output:

```
Ed: Knock knock.
Sam: Who's there?
Ed: Go fmt.
Sam: Go fmt who?
Ed: Go fmt yourself!
```

func NewDecoder

```
func NewDecoder(r io.Reader) *Decoder
```

NewDecoder创建一个从r读取并解码json对象的*Decoder，解码器有自己的缓冲，并可能超前读取部分json数据。

func (*Decoder) Buffered

```
func (dec *Decoder) Buffered() io.Reader
```

Buffered方法返回保存在dec缓存里数据的读取器，该返回值在下次调用Decode方法之前有效。

func (*Decoder) UseNumber

```
func (dec *Decoder) UseNumber()
```

UseNumber方法将dec设置为当接收端是interface{}接口时将json数字解码为Number类型而不是float64类型

func (*Decoder) Decode

```
func (dec *Decoder) Decode(v interface{}) error
```

Decode从输入流读取下一个json编码值并保存在v指向的值里，参见Unmarshal函数的文档获取细节信息。

type Encoder

```
type Encoder struct {
    // 内含隐藏或非导出字段
}
```

Encoder将json对象写入输出流。

func NewEncoder

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder创建一个将数据写入w的*Encoder。

func (*Encoder) Encode

```
func (enc *Encoder) Encode(v interface{}) error
```

Encode将v的json编码写入输出流，并会写入一个换行符，参见Marshal函数的文档获取细节信息。

package pem

```
import "encoding/pem"
```

pem包实现了PEM数据编码（源自保密增强邮件协议）。目前PEM编码主要用于TLS密钥和证书。参见RFC 1421

Index

type Block

```
type Block struct {
    Type    string            // 得自前言的类型（如"RSA PRIVATE KEY"）
    Headers map[string]string // 可选的头项
    Bytes   []byte            // 内容解码后的数据，一般是DER编码的ASN.1
}
```

Block代表PEM编码的结构。编码格式如下：

```
-----BEGIN Type-----
Headers
base64-encoded Bytes
-----END Type-----
```

其中Headers是可为空的多行键值对。

func Decode

```
func Decode(data []byte) (p *Block, rest []byte)
```

Decode函数会从输入里查找到下一个PEM格式的块（证书、私钥等）。它返回解码得到的Block和剩余未解码的数据。如果未发现PEM数据，返回(nil, data)。

func Encode

```
func Encode(out io.Writer, b *Block) error
```

func EncodeToMemory

```
func EncodeToMemory(b *Block) []byte
```

package xml

```
import "encoding/xml"
```

Package xml implements a simple XML 1.0 parser that understands XML name spaces.

Index

Examples

Constants

```
const (
    // 适用于本包Marshal输出的一般性XML header
    // 本常数并不会自动添加到本包的输出里，这里提供主要是出于便利的目的
    Header = `<?xml version="1.0" encoding="UTF-8"?>` + "\n"
)
```

Variables

```
var HTMLAutoClose = htmlAutoClose
```

HTMLAutoClose是应当考虑到自动关闭的HTML元素的集合。

```
var HTMLEntity = htmlEntity
```

HTMLEntity是标准HTML entity字符到其翻译的映射。

type SyntaxError

```
type SyntaxError struct {
    Msg  string
    Line int
}
```

SyntaxError代表XML输入流的格式错误。

func (*SyntaxError) Error

```
func (e *SyntaxError) Error() string
```

type TagPathError

```
type TagPathError struct {
    Struct       reflect.Type
    Field1, Tag1 string
    Field2, Tag2 string
}
```

反序列化时，如果字段标签的路径有冲突，就会返回TagPathError。

func (*TagPathError) Error

```
func (e *TagPathError) Error() string
```

type UnsupportedTypeError

```
type UnsupportedTypeError struct {
    Type reflect.Type
}
```

当序列化时，如果遇到不能转化为XML的类型，就会返回UnsupportedTypeError。

func (*UnsupportedTypeError) Error

```
func (e *UnsupportedTypeError) Error() string
```

type UnmarshalError

```
type UnmarshalError string
```

UnmarshalError代表反序列化时出现的错误。

func (UnmarshalError) Error

```
func (e UnmarshalError) Error() string
```

type CharData

```
type CharData []byte
```

CharData类型代表XML字符数据（原始文本），其中XML转义序列已经被它们所代表的字符取代。

func (CharData) Copy

```
func (c CharData) Copy() CharData
```

type Comment

```
type Comment []byte
```

Comment代表XML注释，格式为<!--comment-->，切片中不包含注释标记<!—和-->。

func (Comment) Copy

```
func (c Comment) Copy() Comment
```

type Directive

```
type Directive []byte
```

Directive代表XML指示，格式为<!directive>，切片中不包含标记<!和>。

func (Directive) Copy

```
func (d Directive) Copy() Directive
```

type ProcInst

```
type ProcInst struct {
    Target string
    Inst   []byte
}
```

ProcInst代表XML处理指令，格式为<?target inst?>。

func (ProcInst) Copy

```
func (p ProcInst) Copy() ProcInst
```

type Name

```
type Name struct {
    Space, Local string
}
```

Name代表一个XML名称（Local字段），并指定名字空间（Space）。Decoder.Token方法返回的Token中，Space标识符是典型的URL而不是被解析的文档里的短前缀。

type Attr

```
type Attr struct {
    Name  Name
    Value string
}
```

Attr代表一个XML元素的一条属性（Name=Value）

type StartElement

```
type StartElement struct {
    Name Name
    Attr []Attr
}
```

StartElement代表一个XML起始元素。

func (StartElement) Copy

```
func (e StartElement) Copy() StartElement
```

func (StartElement) End

```
func (e StartElement) End() EndElement
```

返回e对应的XML结束元素。

type EndElement

```
type EndElement struct {
    Name Name
}
```

EndElement代表一个XML结束元素。

type Token

```
type Token interface{}
```

Token接口用于保存token类型（CharData、Comment、Directive、ProcInst、StartElement、EndElement）的值。

func CopyToken

```
func CopyToken(t Token) Token
```

CopyToken返回一个Token的拷贝。

type Marshaler

```
type Marshaler interface {
    MarshalXML(e *Encoder, start StartElement) error
}
```

实现了Marshaler接口的类型可以将自身序列化为合法的XML元素。

MarshalXML方法将自身调用者编码为零或多个XML元素。 按照惯例，数组或切片会编码为一系列元素，每个成员一条。使用start作为元素标签并不是必须的，但这么做可以帮助Unmarshal方法正确的匹配XML元素和结构体字段。一个常用的策略是在同一个层次里将每个独立的值对应到期望的XML然后使用e.EncodeElement进行编码。另一个常用的策略是重复调用e.EncodeToken来一次一个token的生成XML输出。编码后的token必须组成零或多个XML元素。

type Unmarshaler

```
type Unmarshaler interface {
    UnmarshalXML(d *Decoder, start StartElement) error
}
```

实现了Unmarshaler接口的类型可以根据自身的XML元素描述反序列化自身。

UnmarshalXML方法解码以start起始单个XML元素。如果它返回了错误，外层Unmarshal的调用将停止执行并返回该错误。UnmarshalXML方法必须正好“消费”一个XML元素。一个常用的策略是使用d.DecodeElement 将XML分别解码到各独立值，然后再将这些值写入UnmarshalXML的调用者。另一个常用的策略是使用d.Token一次一个token的处理XML对象。UnmarshalXML通常不使用d.RawToken。

type MarshalerAttr

```
type MarshalerAttr interface {
    MarshalXMLAttr(name Name) (Attr, error)
}
```

实现了MarshalerAttr接口的类型可以将自身序列化为合法的XML属性。

MarshalXMLAttr返回一个值为方法调用者编码后的值的XML属性。使用name作为属性的name并非必须的，但这么做可以帮助Unmarshal方法正确的匹配属性和结构体字段。如果MarshalXMLAttr返回一个零值属性Attr{}，将不会生成属性输出。MarshalXMLAttr只用于有标签且标签有"attr"选项的结构体字段。

type UnmarshalerAttr

```
type UnmarshalerAttr interface {
    UnmarshalXMLAttr(attr Attr) error
}
```

实现了UnmarshalerAttr接口的类型可以根据自身的XML属性形式的描述反序列化自身。

UnmarshalXMLAttr解码单个的XML属性。如果它返回一个错误，外层的Umarshal调用会停止执行并返回该错误。UnmarshalXMLAttr只有在结构体字段的标签有"attr"选项时才被使用。

func Escape

```
func Escape(w io.Writer, s []byte)
```

Escape类似EscapeText函数但会忽略返回的错误。本函数是用于保证和Go 1.0的向后兼容。应用于Go 1.1及以后版本的代码请使用EscapeText。

func EscapeText

```
func EscapeText(w io.Writer, s []byte) error
```

EscapeText向w中写入经过适当转义的、有明文s具有相同意义的XML文本。

func Marshal

```
func Marshal(v interface{}) ([]byte, error)
```

Marshal函数返回v的XML编码。

Marshal处理数组或者切片时会序列化每一个元素。Marshal处理指针时，会序列化其指向的值；如果指针为nil，则啥也不输出。Marshal处理接口时，会序列化其内包含的具体类型值，如果接口值为nil，也是不输出。Marshal处理其余类型数据时，会输出一或多个包含数据的XML元素。

XML元素的名字按如下优先顺序获取：

```
- 如果数据是结构体，其XMLName字段的标签
- 类型为xml.Name的XMLName字段的值
- 数据是某结构体的字段，其标签
- 数据是某结构体的字段，其字段名
- 被序列化的类型的名字
```

一个结构体的XML元素包含该结构体所有导出字段序列化后的元素，有如下例外：

```
- XMLName字段，如上所述，会省略
- 具有标签"-"的字段会省略
- 具有标签"name,attr"的字段会成为该XML元素的名为name的属性
- 具有标签",attr"的字段会成为该XML元素的名为字段名的属性
- 具有标签",chardata"的字段会作为字符数据写入，而非XML元素
- 具有标签",innerxml"的字段会原样写入，而不会经过正常的序列化过程
- 具有标签",comment"的字段作为XML注释写入，而不经过正常的序列化过程，该字段内
- 标签中包含"omitempty"选项的字段如果为空值会省略
  空值为false、0、nil指针、nil接口、长度为0的数组、切片、映射
- 匿名字段（其标签无效）会被处理为其字段是外层结构体的字段
```

如果一个字段的标签为"a>b>c"，则元素c将会嵌套进其上层元素a和b中。如果该字段相邻的字段标签指定了同样的上层元素，则会放在同一个XML元素里。

参见MarshalIndent的例子。如果要求Marshal序列化通道、函数或者映射会返回错误。

func MarshalIndent

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, e
```

MarshalIndent功能类似Marshal。但每个XML元素会另起一行并缩进，该行以prefix起始，后跟一或多个indent的拷贝（根据嵌套层数）。

Example

```
type Address struct {
    City, State string
}
type Person struct {
    XMLName   xml.Name `xml:"person"`
    Id        int      `xml:"id,attr"`
    FirstName string   `xml:"name>first"`
    LastName  string   `xml:"name>last"`
    Age       int      `xml:"age"`
    Height    float32  `xml:"height,omitempty"`
    Married   bool
    Address
    Comment string `xml:",comment"`
}
v := &Person{Id: 13, FirstName: "John", LastName: "Doe", Age: 42}
v.Comment = " Need more details. "
v.Address = Address{"Hanga Roa", "Easter Island"}
output, err := xml.MarshalIndent(v, "  ", "    ")
if err != nil {
    fmt.Printf("error: %v\n", err)
}
os.Stdout.Write(output)
```

Output:

```
  <person id="13">
      <name>
          <first>John</first>
          <last>Doe</last>
      </name>
      <age>42</age>
      <Married>false</Married>
      <City>Hanga Roa</City>
      <State>Easter Island</State>
      <!-- Need more details. -->
  </person>
```

func Unmarshal

```
func Unmarshal(data []byte, v interface{}) error
```

Unmarshal解析XML编码的数据并将结果存入v指向的值。v只能指向结构体、切片或者和字符串。良好格式化的数据如果不能存入v，会被丢弃。

因为Unmarshal使用reflect包，它只能填写导出字段。本函数好似用大小写敏感的比较来匹配XML元素名和结构体的字段名/标签键名。

Unmarshal函数使用如下规则将XML元素映射到结构体字段上。这些规则中，字段标签指的是结构体字段的标签键'xml'对应的值（参见上面的例子）：

```
* 如果结构体字段的类型为字符串或者[]byte，且标签为",innerxml",
  Unmarshal函数直接将对应原始XML文本写入该字段，其余规则仍适用。
* 如果结构体字段类型为xml.Name且名为XMLName，Unmarshal会将元素名写入该字段
* 如果字段XMLName的标签的格式为"name"或"namespace-URL name",
  XML元素必须有给定的名字（以及可选的名字空间），否则Unmarshal会返回错误。
* 如果XML元素的属性的名字匹配某个标签",attr"为字段的字段名，或者匹配某个标签为
  的字段的标签名，Unmarshal会将该属性的值写入该字段。
* 如果XML元素包含字符数据，该数据会存入结构体中第一个具有标签",chardata"的字
  该字段可以是字符串类型或者[]byte类型。如果没有这样的字段，字符数据会丢弃。
* 如果XML元素包含注释，该数据会存入结构体中第一个具有标签",comment"的字段中，
  该字段可以是字符串类型或者[]byte类型。如果没有这样的字段，字符数据会丢弃。
* 如果XML元素包含一个子元素，其名称匹配格式为"a"或"a>b>c"的标签的前缀，反序列
  XML结构中寻找具有指定名称的元素，并将最后端的元素映射到该标签所在的结构体字段
  以">"开始的标签等价于以字段名开始并紧跟着">" 的标签。
* 如果XML元素包含一个子元素，其名称匹配某个结构体类型字段的XMLName字段的标签名
  且该结构体字段本身没有显式指定标签名，Unmarshal会将该元素映射到该字段。
* 如果XML元素的包含一个子元素，其名称匹配够格结构体字段的字段名，且该字段没有任
  （",attr"、",chardata"等），Unmarshal会将该元素映射到该字段。
* 如果XML元素包含的某个子元素不匹配以上任一条，而存在某个字段其标签为",any",
  Unmarshal会将该元素映射到该字段。
* 匿名字段被处理为其字段好像位于外层结构体中一样。
* 标签为"-"的结构体字段永不会被反序列化填写。
```

Unmarshal函数将XML元素写入string或[]byte时，会将该元素的字符数据串联起来作为值，目标[]byte不能是nil。

Unmarshal函数将属性写入string或[]byte时，会将属性的值以字符串/切片形式写入。

Unmarshal函数将XML元素写入切片时，会将切片扩展并将XML元素的子元素映射入新建的值里。

Unmarshal函数将XML元素/属性写入bool值时，会将对应的字符串转化为布尔值。

Unmarshal函数将XML元素/属性写入整数或浮点数类型时，会将对应的字符串解释为十进制数字。不会检查溢出。

Unmarshal函数将XML元素写入xml.Name类型时，会记录元素的名称。

Unmarshal函数将XML元素写入指针时，会申请一个新值并将XML元素映射入该值。

Example

```
type Email struct {
    Where string `xml:"where,attr"`
    Addr  string
}
type Address struct {
    City, State string
}
type Result struct {
    XMLName xml.Name `xml:"Person"`
    Name    string   `xml:"FullName"`
    Phone   string
    Email   []Email
    Groups  []string `xml:"Group>Value"`
    Address
}
v := Result{Name: "none", Phone: "none"}
data := `
        <Person>
            <FullName>Grace R. Emlin</FullName>
            <Company>Example Inc.</Company>
            <Email where="home">
                <Addr>gre@example.com</Addr>
            </Email>
            <Email where='work'>
                <Addr>gre@work.com</Addr>
            </Email>
            <Group>
                <Value>Friends</Value>
                <Value>Squash</Value>
            </Group>
            <City>Hanga Roa</City>
            <State>Easter Island</State>
        </Person>
    `
err := xml.Unmarshal([]byte(data), &v)
if err != nil {
    fmt.Printf("error: %v", err)
    return
}
fmt.Printf("XMLName: %#v\n", v.XMLName)
fmt.Printf("Name: %q\n", v.Name)
fmt.Printf("Phone: %q\n", v.Phone)
fmt.Printf("Email: %v\n", v.Email)
fmt.Printf("Groups: %v\n", v.Groups)
fmt.Printf("Address: %v\n", v.Address)
```

Output:

```
XMLName: xml.Name{Space:"", Local:"Person"}
Name: "Grace R. Emlin"
Phone: "none"
Email: [{home gre@example.com} {work gre@work.com}]
Groups: [Friends Squash]
Address: {Hanga Roa Easter Island}
```

type Decoder

```
type Decoder struct {
    // Strict默认设为true，强制要求符合XML规范
    // 如果设为false，则解析器允许输入中包含常见的错误：
    //   * 如果元素缺少结束标签，解析器会虚构一个结束标签以保证返回值来自良好
    //   * 属性值和字符数据中，未知或畸形的字符entity（以&开始的序列）会丢在
    //
    // 设置：
    //
    //   d.Strict = false
    //   d.AutoClose = HTMLAutoClose
    //   d.Entity = HTMLEntity
    //
    // 可以创建一个能处理标准HTML的解析器。
    //
    // Strict模式不会强制要求XML名称空间TR，特别注意它不会拒绝使用未定义前缀
    // 这些标签会将未知前缀作为名字空间URL来记录
    Strict bool
    // 当Strict == false时，AutoClose指定一个元素的集合：
    // 这些元素在开启后就立刻结束，不管有没有对应关闭标签存在
    AutoClose []string
    // Entity字段用于将非标准的实体名映射到替换的字符串
    // parser的行为就好像标准实体映射存在于本字段，即使实际上本字段没有：
    //
    //    "lt": "<",
    //    "gt": ">",
    //    "amp": "&",
    //    "apos": "'",
    //    "quot": `"`,
    Entity map[string]string
    // CharsetReader字段如果非nil，会定义一个函数来生成转换字符集的io.Reac
    // 将给定的非utf-8字符集转换为utf-8字符集。如果CharsetReader字段为nil
    // 或者返回一个错误，解析将会停止并发挥该错误。CharsetReader的返回值不能
    CharsetReader func(charset string, input io.Reader) (io.Reader,
    // DefaultSpace设置未修饰标签的默认名字空间，就好像整个XML流都包进有个
    // xmlns="DefaultSpace"的元素内
    DefaultSpace string
    // 内含隐藏或非导出字段
}
```

Decoder代表一个XML解析器，可以读取输入流的部分数据，该解析器假定输入是utf-8编码的。

func NewDecoder

```
func NewDecoder(r io.Reader) *Decoder
```

创建一个从r读取XML数据的解析器。如果r未实现io.ByteReader接口，NewDecoder会为其添加缓存。

func (*Decoder) Decode

```
func (d *Decoder) Decode(v interface{}) error
```

Decode方法功能类似xml.Unmarshal函数，但会从底层读取XML数据并查找StartElement。

func (*Decoder) DecodeElement

```
func (d *Decoder) DecodeElement(v interface{}, start *StartElement)
```

DecodeElement方法的功能类似xml.Unmarshal函数，但它会启出一个指向XML起始标签后将解析结果写入v。当客户端自己读取了一些原始XML token但仍想defer调用Unmarshal处理一些元素时很有用。

func (*Decoder) Token

```
func (d *Decoder) Token() (t Token, err error)
```

Token返回输入流里的下一个XML token。在输入流的结尾处，会返回(nil, io.EOF)

返回的token数据里的[]byte数据引用自解析器内部的缓存，只在下一次调用Token之前有效。如要获取切片的拷贝，调用CopyToken函数或者token的Copy方法。

成功调用的Token方法会将自我闭合的元素（如
）扩展为分离的起始和结束标签。

Token方法会保证它返回的StartElement和EndElement两种token正确的嵌套和匹配：如果本方法遇到了不正确的结束标签，会返回一个错误。

Token方法实现了XML名字空间，细节参见http://www.w3.org/TR/REC-xml-names/。每一个包含在Token里的Name结构体，都会将Space字段设为URL标识（如果可知的话）。如果Token遇到未知的名字空间前缀，它会使用该前缀作为名字空间，而不是报错。

func (*Decoder) RawToken

```
func (d *Decoder) RawToken() (Token, error)
```

RawToken方法Token方法，但不会验证起始和结束标签，也不将名字空间前缀翻译为它们相应的URL。

func (*Decoder) Skip

```
func (d *Decoder) Skip() error
```

Skip从底层读取token，直到读取到最近一次读取到的起始标签对应的结束标签。如果读取中遇到别的起始标签会进行迭代，因此可以跳过嵌套结构。如果本方法找到了对应起始标签的结束标签，会返回nil；否则返回一个描述该问题的错误。

type Encoder

```
type Encoder struct {
    // 内含隐藏或非导出字段
}
```

Encoder向输出流中写入XML数据。

Example

```
type Address struct {
    City, State string
}
type Person struct {
    XMLName   xml.Name `xml:"person"`
    Id        int      `xml:"id,attr"`
    FirstName string   `xml:"name>first"`
    LastName  string   `xml:"name>last"`
    Age       int      `xml:"age"`
    Height    float32  `xml:"height,omitempty"`
    Married   bool
    Address
    Comment string `xml:",comment"`
}
v := &Person{Id: 13, FirstName: "John", LastName: "Doe", Age: 42}
v.Comment = " Need more details. "
v.Address = Address{"Hanga Roa", "Easter Island"}
enc := xml.NewEncoder(os.Stdout)
enc.Indent("  ", "    ")
if err := enc.Encode(v); err != nil {
    fmt.Printf("error: %v\n", err)
}
```

Output:

```
  <person id="13">
      <name>
          <first>John</first>
          <last>Doe</last>
      </name>
      <age>42</age>
      <Married>false</Married>
      <City>Hanga Roa</City>
      <State>Easter Island</State>
      <!-- Need more details. -->
  </person>
```

func NewEncoder

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder创建一个写入w的*Encoder。

func (*Encoder) Encode

```
func (enc *Encoder) Encode(v interface{}) error
```

Encode将v编码为XML后写入底层。参见Marshal函数获取go到XML转换的细节。在返回前enc会调用Flush。

func (*Encoder) EncodeElement

```
func (enc *Encoder) EncodeElement(v interface{}, start StartElement
```

EncodeElement将v的XML编码写入底层， 并使用start作为编码的最外层。参见Marshal函数获取go到XML转换的细节。在返回前enc会调用Flush方法。

func (*Encoder) EncodeToken

```
func (enc *Encoder) EncodeToken(t Token) error
```

EncodeToken向底层写入一个token。如果StartElement和EndElement的匹配不正确，本方法会返回错误。

EncodeToken 方法不会调用Flush，因为它通常是更大型操作如Encode或EncodeElement方法的一部分（或者用户自定义的Marshaler接口MarshalXML 方法里调用本方法），这些方法会在结束前Flush。调用者创建一个Encoder并直接使用本方法而不使用Encode或EncodeElement方法的话，必须在结束时调用Flush以保证XML数据写入底层的io.Writer接口。

EncodeToken写入ProcInst类型Token时，只允许在底层流最开始写入目标是"xml"的ProcInst。

func (*Encoder) Flush

```
func (enc *Encoder) Flush() error
```

Flush方法会将缓存中的XML数据写入底层。参见EncodeToken函数获取细节信息。

func (*Encoder) Indent

```
func (enc *Encoder) Indent(prefix, indent string)
```

Indent函数设定编码器生成XML数据时的格式化缩进信息。细节请参见MarshalIndent函数。

Bugs

☞ XML元素和数据结构体的映射有天生的缺陷：XML元素是依赖顺序的匿名值的集合，而结构体是不依赖顺序的命名值的集合。参见json包获取更适用于数据结构体的文本表示。

package errors

```
import "errors"
```

errors包实现了创建错误值的函数。

Example

```
package errors_test
import (
    "fmt"
    "time"
)
// MyError is an error implementation that includes a time and mess
type MyError struct {
    When time.Time
    What string
}
func (e MyError) Error() string {
    return fmt.Sprintf("%v: %v", e.When, e.What)
}
func oops() error {
    return MyError{
        time.Date(1989, 3, 15, 22, 30, 0, 0, time.UTC),
        "the file system has gone away",
    }
}
func Example() {
    if err := oops(); err != nil {
        fmt.Println(err)
    }
    // Output: 1989-03-15 22:30:00 +0000 UTC: the file system has g
}
```

Index

Examples

func New

```
func New(text string) error
```

使用字符串创建一个错误,请类比fmt包的Errorf方法，差不多可以认为是New(fmt.Sprintf(...))。

Example

```
err := errors.New("emit macho dwarf: elf header corrupted")
if err != nil {
    fmt.Print(err)
}
```

Output:

```
emit macho dwarf: elf header corrupted
```

Example (Errorf)

```
const name, id = "bimmler", 17
err := fmt.Errorf("user %q (id %d) not found", name, id)
if err != nil {
    fmt.Print(err)
}
```

Output:

```
user "bimmler" (id 17) not found
```

package expvar

```
import "expvar"
```

expvar包提供了公共变量的标准接口，如服务的操作计数器。本包通过HTTP在/debug/vars位置以JSON格式导出了这些变量。

对这些公共变量的读写操作都是原子级的。

为了增加HTTP处理器，本包注册了如下变量：

```
cmdline   os.Args
memstats  runtime.Memstats
```

有时候本包被导入只是为了获得本包注册HTTP处理器和上述变量的副作用。此时可以如下方式导入本包：

```
import _ "expvar"
```

Index

type Var

```
type Var interface {
    String() string
}
```

Var接口是所有导出变量的抽象类型。

type Int

```
type Int struct {
    // 内含隐藏或非导出字段
}
```

Int代表一个64位整数变量，满足Var接口。

func NewInt

```
func NewInt(name string) *Int
```

func (*Int) Add

```
func (v *Int) Add(delta int64)
```

func (*Int) Set

```
func (v *Int) Set(value int64)
```

func (*Int) String

```
func (v *Int) String() string
```

type Float

```
type Float struct {
    // 内含隐藏或非导出字段
}
```

Float代表一个64位浮点数变量，满足Var接口。

func NewFloat

```
func NewFloat(name string) *Float
```

func (*Float) Add

```
func (v *Float) Add(delta float64)
```

Add adds delta to v.

func (*Float) Set

```
func (v *Float) Set(value float64)
```

Set sets v to value.

func (*Float) String

```
func (v *Float) String() string
```

type String

```
type String struct {
    // 内含隐藏或非导出字段
}
```

String代表一个字符串变量，满足Var接口。

func NewString

```
func NewString(name string) *String
```

func (*String) Set

```
func (v *String) Set(value string)
```

func (*String) String

```
func (v *String) String() string
```

type Func

```
type Func func() interface{}
```

Func通过调用函数并将结果编码为json，实现了Var接口。

func (Func) String

```
func (f Func) String() string
```

type KeyValue

```
type KeyValue struct {
    Key   string
    Value Var
}
```

KeyValue代表Map中的一条记录。（键值对）

type Map

```
type Map struct {
    // 内含隐藏或非导出字段
}
```

Map代表一个string到Var的映射变量，满足Var接口。

func NewMap

```
func NewMap(name string) *Map
```

func (*Map) Init

```
func (v *Map) Init() *Map
```

func (*Map) Get

```
func (v *Map) Get(key string) Var
```

func (*Map) Set

```
func (v *Map) Set(key string, av Var)
```

func (*Map) Add

```
func (v *Map) Add(key string, delta int64)
```

func (*Map) AddFloat

```
func (v *Map) AddFloat(key string, delta float64)
```

AddFloat向索引key对应的值（底层为*Float）修改为加上delta后的值。

func (*Map) Do

```
func (v *Map) Do(f func(KeyValue))
```

Do对映射的每一条记录都调用f。迭代执行时会锁定该映射，但已存在的记录可以同时更新。

func (*Map) String

```
func (v *Map) String() string
```

func Get

```
func Get(name string) Var
```

Get获取名为name的导出变量。

func Publish

```
func Publish(name string, v Var)
```

Publish声明一个导出变量。必须在init函数里调用。如果name已经被注册，会调用log.Panic。

func Do

```
func Do(f func(KeyValue))
```

Do对导出变量的每一条记录都调用f。迭代执行时会锁定全局变量映射，但已存在的记录可以同时更新。

package flag

```
import "flag"
```

flag包实现了命令行参数的解析。

要求：

使用flag.String(), Bool(), Int()等函数注册flag，下例声明了一个整数flag，解析结果保存在*int指针ip里：

```
import "flag"
var ip = flag.Int("flagname", 1234, "help message for flagname")
```

如果你喜欢，也可以将flag绑定到一个变量，使用Var系列函数：

```
var flagvar int
func init() {
    flag.IntVar(&flagvar, "flagname", 1234, "help message for flagr
}
```

或者你可以自定义一个用于flag的类型（满足Value接口）并将该类型用于flag解析，如下：

```
flag.Var(&flagVal, "name", "help message for flagname")
```

对这种flag，默认值就是该变量的初始值。

在所有flag都注册之后，调用：

```
flag.Parse()
```

来解析命令行参数写入注册的flag里。

解析之后，flag的值可以直接使用。如果你使用的是flag自身，它们是指针；如果你绑定到了某个变量，它们是值。

```
fmt.Println("ip has value ", *ip)
fmt.Println("flagvar has value ", flagvar)
```

解析后，flag后面的参数可以从flag.Args()里获取或用flag.Arg(i)单独获取。这些参数的索引为从0到flag.NArg()-1。

命令行flag语法：

```
-flag
-flag=x
-flag x  // 只有非bool类型的flag可以
```

可以使用1个或2个'-'号，效果是一样的。最后一种格式不能用于bool类型的flag，因为如果有文件名为0、false等时,如下命令：

```
cmd -x *
```

其含义会改变。你必须使用-flag=false格式来关闭一个bool类型flag。

Flag解析在第一个非flag参数（单个"-"不是flag参数）之前停止，或者在终止符"--"之后停止。

整数flag接受1234、0664、0x1234等类型，也可以是负数。bool类型flag可以是：

```
1, 0, t, f, T, F, true, false, TRUE, FALSE, True, False
```

时间段flag接受任何合法的可提供给time.ParseDuration的输入。

默认的命令行flag集被包水平的函数控制。FlagSet类型允许程序员定义独立的flag集，例如实现命令行界面下的子命令。FlagSet的方法和包水平的函数是非常类似的。

Example

```
// These examples demonstrate more intricate uses of the flag packa
package flag_test
import (
    "errors"
    "flag"
    "fmt"
    "strings"
    "time"
)
// Example 1: A single string flag called "species" with default va
var species = flag.String("species", "gopher", "the species we are
// Example 2: Two flags sharing a variable, so we can have a shorth
// The order of initialization is undefined, so make sure both use
// same default value. They must be set up with an init function.
var gopherType string
func init() {
    const (
```

```
        defaultGopher = "pocket"
        usage         = "the variety of gopher"
    )
    flag.StringVar(&gopherType, "gopher_type", defaultGopher, usage
    flag.StringVar(&gopherType, "g", defaultGopher, usage+" (shorth
}
// Example 3: A user-defined flag type, a slice of durations.
type interval []time.Duration
// String is the method to format the flag's value, part of the fla
// The String method's output will be used in diagnostics.
func (i *interval) String() string {
    return fmt.Sprint(*i)
}
// Set is the method to set the flag value, part of the flag.Value
// Set's argument is a string to be parsed to set the flag.
// It's a comma-separated list, so we split it.
func (i *interval) Set(value string) error {
    // If we wanted to allow the flag to be set multiple times,
    // accumulating values, we would delete this if statement.
    // That would permit usages such as
    //    -deltaT 10s -deltaT 15s
    // and other combinations.
    if len(*i) > 0 {
        return errors.New("interval flag already set")
    }
    for _, dt := range strings.Split(value, ",") {
        duration, err := time.ParseDuration(dt)
        if err != nil {
            return err
        }
        *i = append(*i, duration)
    }
    return nil
}
// Define a flag to accumulate durations. Because it has a special
// we need to use the Var function and therefore create the flag du
// init.
var intervalFlag interval
func init() {
    // Tie the command-line flag to the intervalFlag variable and
    // set a usage message.
    flag.Var(&intervalFlag, "deltaT", "comma-separated list of inte
}
func Example() {
    // All the interesting pieces are with the variables declared a
    // to enable the flag package to see the flags defined there, o
    // execute, typically at the start of main (not init!):
    //    flag.Parse()
    // We don't run it here because this is not a main function and
    // the testing suite has already parsed the flags.
}
```

Index

Examples

Variables

```
var CommandLine = NewFlagSet(os.Args[0], ExitOnError)
```

CommandLine是默认的命令行flag集，用于解析os.Args。包水平的函数如BoolVar、Arg等都是对其方法的封装。

```
var ErrHelp = errors.New("flag: help requested")
```

当flag -help被调用但没有注册这个flag时，就会返回ErrHelp。

```
var Usage = func() {
    fmt.Fprintf(os.Stderr, "Usage of %s:\n", os.Args[0])
    PrintDefaults()
}
```

Usage打印到标准错误输出一个使用信息，记录了所有注册的flag。本函数是一个包变量，可以将其修改为指向自定义的函数。

type Value

```
type Value interface {
    String() string
    Set(string) error
}
```

Value接口是用于将动态的值保存在一个flag里。（默认值被表示为一个字符串）

如果Value接口具有IsBoolFlag() bool方法，且返回真。命令行解析其会将-name等价为-name=true而不是使用下一个命令行参数。

type Getter

```
type Getter interface {
    Value
    Get() interface{}
}
```

Gette接口使可以取回Value接口的内容。本接口包装了Value接口而不是作为Value接口的一部分，因为本接口是在Go 1之后出现，出于兼容的考虑才如此。本包所有的满足Value接口的类型都同时满足Getter接口。

type ErrorHandling

```
type ErrorHandling int
```

ErrorHandling定义如何处理flag解析错误。

```
const (
    ContinueOnError ErrorHandling = iota
    ExitOnError
    PanicOnError
)
```

type Flag

```
type Flag struct {
    Name     string // flag在命令行中的名字
    Usage    string // 帮助信息
    Value    Value  // 要设置的值
    DefValue string // 默认值（文本格式），用于使用信息
}
```

Flag类型代表一条flag的状态。

type FlagSet

```
type FlagSet struct {
    // Usage函数在解析flag出现错误时会被调用
    // 该字段为一个函数（而非采用方法），以便修改为自定义的错误处理函数
    Usage func()
    // 内含隐藏或非导出字段
}
```

FlagSet代表一个已注册的flag的集合。FlagSet零值没有名字，采用ContinueOnError错误处理策略。

func NewFlagSet

```
func NewFlagSet(name string, errorHandling ErrorHandling) *FlagSet
```

NewFlagSet创建一个新的、名为name，采用errorHandling为错误处理策略的FlagSet。

func (*FlagSet) Init

```
func (f *FlagSet) Init(name string, errorHandling ErrorHandling)
```

Init设置flag集合f的名字和错误处理属性。FlagSet零值没有名字，默认采用ContinueOnError错误处理策略。

func (*FlagSet) NFlag

```
func (f *FlagSet) NFlag() int
```

NFlag返回解析时进行了设置的flag的数量。

func (*FlagSet) Lookup

```
func (f *FlagSet) Lookup(name string) *Flag
```

返回已经f中已注册flag的Flag结构体指针；如果flag不存在的话，返回nil。

func (*FlagSet) NArg

```
func (f *FlagSet) NArg() int
```

NArg返回解析flag之后剩余参数的个数。

func (*FlagSet) Args

```
func (f *FlagSet) Args() []string
```

返回解析之后剩下的非flag参数。（不包括命令名）

func (*FlagSet) Arg

```
func (f *FlagSet) Arg(i int) string
```

返回解析之后剩下的第i个参数，从0开始索引。

func (*FlagSet) PrintDefaults

```
func (f *FlagSet) PrintDefaults()
```

PrintDefault打印集合中所有注册好的flag的默认值。除非另外配置，默认输出到标准错误输出中。

func (*FlagSet) SetOutput

```
func (f *FlagSet) SetOutput(output io.Writer)
```

设置使用信息和错误信息的输出流，如果output为nil，将使用os.Stderr。

func (*FlagSet) Bool

```
func (f *FlagSet) Bool(name string, value bool, usage string) *boo
```

Bool用指定的名称、默认值、使用信息注册一个bool类型flag。返回一个保存了该flag的值的指针。

func (*FlagSet) BoolVar

```
func (f *FlagSet) BoolVar(p *bool, name string, value bool, usage s
```

BoolVar用指定的名称、默认值、使用信息注册一个bool类型flag，并将flag的值保存到p指向的变量。

func (*FlagSet) Int

```
func (f *FlagSet) Int(name string, value int, usage string) *int
```

Int用指定的名称、默认值、使用信息注册一个int类型flag。返回一个保存了该flag的值的指针。

func (*FlagSet) IntVar

```
func (f *FlagSet) IntVar(p *int, name string, value int, usage stri
```

IntVar用指定的名称、默认值、使用信息注册一个int类型flag，并将flag的值保存到p指向的变量。

func (*FlagSet) Int64

```
func (f *FlagSet) Int64(name string, value int64, usage string) *in
```

Int64用指定的名称、默认值、使用信息注册一个int64类型flag。返回一个保存了该flag的值的指针。

func (*FlagSet) Int64Var

```
func (f *FlagSet) Int64Var(p *int64, name string, value int64, usag
```

Int64Var用指定的名称、默认值、使用信息注册一个int64类型flag，并将flag的值保存到p指向的变量。

func (*FlagSet) Uint

```
func (f *FlagSet) Uint(name string, value uint, usage string) *uint
```

Uint用指定的名称、默认值、使用信息注册一个uint类型flag。返回一个保存了该flag的值的指针。

func (*FlagSet) UintVar

```
func (f *FlagSet) UintVar(p *uint, name string, value uint, usage s
```

UintVar用指定的名称、默认值、使用信息注册一个uint类型flag，并将flag的值保存到p指向的变量。

func (*FlagSet) Uint64

```
func (f *FlagSet) Uint64(name string, value uint64, usage string) *
```

Uint64用指定的名称、默认值、使用信息注册一个uint64类型flag。返回一个保存了该flag的值的指针。

func (*FlagSet) Uint64Var

```
func (f *FlagSet) Uint64Var(p *uint64, name string, value uint64, u
```

Uint64Var用指定的名称、默认值、使用信息注册一个uint64类型flag，并将flag的值保存到p指向的变量。

func (*FlagSet) Float64

```
func (f *FlagSet) Float64(name string, value float64, usage string)
```

Float64用指定的名称、默认值、使用信息注册一个float64类型flag。返回一个保存了该flag的值的指针。

func (*FlagSet) Float64Var

```
func (f *FlagSet) Float64Var(p *float64, name string, value float64
```

Float64Var用指定的名称、默认值、使用信息注册一个float64类型flag，并将flag的值保存到p指向的变量。

func (*FlagSet) String

```
func (f *FlagSet) String(name string, value string, usage string) *
```

String用指定的名称、默认值、使用信息注册一个string类型flag。返回一个保存了该flag的值的指针。

func (*FlagSet) StringVar

```
func (f *FlagSet) StringVar(p *string, name string, value string, u
```

StringVar用指定的名称、默认值、使用信息注册一个string类型flag，并将flag的值保存到p指向的变量。

func (*FlagSet) Duration

```
func (f *FlagSet) Duration(name string, value time.Duration, usage
```

Duration用指定的名称、默认值、使用信息注册一个time.Duration类型flag。返回一个保存了该flag的值的指针。

func (*FlagSet) DurationVar

```
func (f *FlagSet) DurationVar(p *time.Duration, name string, value
```

DurationVar用指定的名称、默认值、使用信息注册一个time.Duration类型flag，并将flag的值保存到p指向的变量。

func (*FlagSet) Var

```
func (f *FlagSet) Var(value Value, name string, usage string)
```

Var方法使用指定的名字、使用信息注册一个flag。该flag的类型和值由第一个参数表示，该参数应实现了Value接口。例如，用户可以创建一个flag，可以用Value接口的Set方法将逗号分隔的字符串转化为字符串切片。

func (*FlagSet) Set

```
func (f *FlagSet) Set(name, value string) error
```

设置已注册的flag的值。

func (*FlagSet) Parse

```
func (f *FlagSet) Parse(arguments []string) error
```

从arguments中解析注册的flag。必须在所有flag都注册好而未访问其值时执行。未注册却使用flag -help时，会返回ErrHelp。

func (*FlagSet) Parsed

```
func (f *FlagSet) Parsed() bool
```

返回是否f.Parse已经被调用过。

func (*FlagSet) Visit

```
func (f *FlagSet) Visit(fn func(*Flag))
```

按照字典顺序遍历标签，并且对每个标签调用fn。 这个函数只遍历解析时进行了设置的标签。

func (*FlagSet) VisitAll

```
func (f *FlagSet) VisitAll(fn func(*Flag))
```

按照字典顺序遍历标签，并且对每个标签调用fn。 这个函数会遍历所有标签，不管解析时有无进行设置。

func NFlag

```
func NFlag() int
```

NFlag返回已被设置的flag的数量。

func Lookup

```
func Lookup(name string) *Flag
```

返回已经已注册flag的Flag结构体指针；如果flag不存在的话，返回nil。。

func NArg

```
func NArg() int
```

NArg返回解析flag之后剩余参数的个数。

func Args

```
func Args() []string
```

返回解析之后剩下的非flag参数。（不包括命令名）

func Arg

```
func Arg(i int) string
```

返回解析之后剩下的第i个参数，从0开始索引。

func PrintDefaults

```
func PrintDefaults()
```

PrintDefault会向标准错误输出写入所有注册好的flag的默认值。

func Bool

```
func Bool(name string, value bool, usage string) *bool
```

Bool用指定的名称、默认值、使用信息注册一个bool类型flag。返回一个保存了该flag的值的指针。

func BoolVar

```
func BoolVar(p *bool, name string, value bool, usage string)
```

BoolVar用指定的名称、默认值、使用信息注册一个bool类型flag，并将flag的值保存到p指向的变量。

func Int

```
func Int(name string, value int, usage string) *int
```

Int用指定的名称、默认值、使用信息注册一个int类型flag。返回一个保存了该flag的值的指针。

func IntVar

```
func IntVar(p *int, name string, value int, usage string)
```

IntVar用指定的名称、默认值、使用信息注册一个int类型flag，并将flag的值保存到p指向的变量。

func Int64

```
func Int64(name string, value int64, usage string) *int64
```

Int64用指定的名称、默认值、使用信息注册一个int64类型flag。返回一个保存了该flag的值的指针。

func Int64Var

```
func Int64Var(p *int64, name string, value int64, usage string)
```

Int64Var用指定的名称、默认值、使用信息注册一个int64类型flag，并将flag的值保存到p指向的变量。

func Uint

```
func Uint(name string, value uint, usage string) *uint
```

Uint用指定的名称、默认值、使用信息注册一个uint类型flag。返回一个保存了该flag的值的指针。

func UintVar

```
func UintVar(p *uint, name string, value uint, usage string)
```

UintVar用指定的名称、默认值、使用信息注册一个uint类型flag，并将flag的值保存到p指向的变量。

func Uint64

```
func Uint64(name string, value uint64, usage string) *uint64
```

Uint64用指定的名称、默认值、使用信息注册一个uint64类型flag。返回一个保存了该flag的值的指针。

func Uint64Var

```
func Uint64Var(p *uint64, name string, value uint64, usage string)
```

Uint64Var用指定的名称、默认值、使用信息注册一个uint64类型flag，并将flag的值保存到p指向的变量。

func Float64

```
func Float64(name string, value float64, usage string) *float64
```

Float64用指定的名称、默认值、使用信息注册一个float64类型flag。返回一个保存了该flag的值的指针。

func Float64Var

```
func Float64Var(p *float64, name string, value float64, usage strin
```

Float64Var用指定的名称、默认值、使用信息注册一个float64类型flag，并将flag的值保存到p指向的变量。

func String

```
func String(name string, value string, usage string) *string
```

String用指定的名称、默认值、使用信息注册一个string类型flag。返回一个保存了该flag的值的指针。

func StringVar

```
func StringVar(p *string, name string, value string, usage string)
```

StringVar用指定的名称、默认值、使用信息注册一个string类型flag，并将flag的值保存到p指向的变量。

func Duration

```
func Duration(name string, value time.Duration, usage string) *time
```

Duration用指定的名称、默认值、使用信息注册一个time.Duration类型flag。返回一个保存了该flag的值的指针。

func DurationVar

```
func DurationVar(p *time.Duration, name string, value time.Duration
```

DurationVar用指定的名称、默认值、使用信息注册一个time.Duration类型flag，并将flag的值保存到p指向的变量。

func Var

```
func Var(value Value, name string, usage string)
```

Var方法使用指定的名字、使用信息注册一个flag。该flag的类型和值由第一个参数表示，该参数应实现了Value接口。例如，用户可以创建一个flag，可以用Value接口的Set方法将逗号分隔的字符串转化为字符串切片。

func Set

```
func Set(name, value string) error
```

设置已注册的flag的值。

func Parse

```
func Parse()
```

从os.Args[1:]中解析注册的flag。必须在所有flag都注册好而未访问其值时执行。未注册却使用flag -help时，会返回ErrHelp。

func Parsed

```
func Parsed() bool
```

返回是否Parse已经被调用过。

func Visit

```
func Visit(fn func(*Flag))
```

按照字典顺序遍历标签，并且对每个标签调用fn。 这个函数只遍历解析时进行了设置的标签。

func VisitAll

```
func VisitAll(fn func(*Flag))
```

按照字典顺序遍历标签，并且对每个标签调用fn。 这个函数会遍历所有标签，不管解析时有无进行设置。

package fmt

```
import "fmt"
```

mt包实现了类似C语言printf和scanf的格式化I/O。格式化动作（'verb'）源自C语言但更简单。

Printing

verb：

通用：

```
%v      值的默认格式表示
%+v     类似%v，但输出结构体时会添加字段名
%#v     值的Go语法表示
%T      值的类型的Go语法表示
%%      百分号
```

布尔值：

```
%t      单词true或false
```

整数：

```
%b      表示为二进制
%c      该值对应的unicode码值
%d      表示为十进制
%o      表示为八进制
%q      该值对应的单引号括起来的go语法字符字面值，必要时会采用安全的转义表示
%x      表示为十六进制，使用a-f
%X      表示为十六进制，使用A-F
%U      表示为Unicode格式：U+1234，等价于"U+%04X"
```

浮点数与复数的两个组分：

```
%b    无小数部分、二进制指数的科学计数法，如-123456p-78；参见strconv.Forma
%e    科学计数法，如-1234.456e+78
%E    科学计数法，如-1234.456E+78
%f    有小数部分但无指数部分，如123.456
%F    等价于%f
%g    根据实际情况采用%e或%f格式（以获得更简洁、准确的输出）
%G    根据实际情况采用%E或%F格式（以获得更简洁、准确的输出）
```

字符串和[]byte：

```
%s    直接输出字符串或者[]byte
%q    该值对应的双引号括起来的go语法字符串字面值，必要时会采用安全的转义表示
%x    每个字节用两字符十六进制数表示（使用a-f）
%X    每个字节用两字符十六进制数表示（使用A-F）
```

指针：

```
%p    表示为十六进制，并加上前导的0x
```

没有%u。整数如果是无符号类型自然输出也是无符号的。类似的，也没有必要指定操作数的尺寸（int8，int64）。

宽度通过一个紧跟在百分号后面的十进制数指定，如果未指定宽度，则表示值时除必需之外不作填充。精度通过（可选的）宽度后跟点号后跟的十进制数指定。如果未指定精度，会使用默认精度；如果点号后没有跟数字，表示精度为0。举例如下：

```
%f:     默认宽度，默认精度
%9f     宽度9，默认精度
%.2f    默认宽度，精度2
%9.2f   宽度9，精度2
%9.f    宽度9，精度0
```

宽度和精度格式化控制的是Unicode码值的数量（不同于C的printf，它的这两个因数指的是字节的数量）。两者任一个或两个都可以使用'*'号取代，此时它们的值将被对应的参数（按"*"号和verb出现的顺序，即控制其值的参数会出现在要表示的值前面）控制，这个操作数必须是int类型。

对于大多数类型的值，宽度是输出字符数目的最小数量，如果必要会用空格填充。对于字符串，精度是输出字符数目的最大数量，如果必要会截断字符串。

对于整数，宽度和精度都设置输出总长度。采用精度时表示右对齐并用0填充，而宽度默认表示用空格填充。

对于浮点数，宽度设置输出总长度；精度设置小数部分长度（如果有的话），除了%g和%G，此时精度设置总的数字个数。例如，对数字123.45，格式%6.2f 输出123.45；格式%.4g输出123.5。%e和%f的默认精度是6，%g的默认精度是可以将该值区分出来需要的最小数字个数。

对复数，宽度和精度会分别用于实部和虚部，结果用小括号包裹。因此%f用于1.2+3.4i输出(1.200000+3.400000i)。

其它flag：

```
'+'     总是输出数值的正负号；对%q（%+q）会生成全部是ASCII字符的输出（通过转义
' '     对数值，正数前加空格而负数前加负号；
'-'     在输出右边填充空白而不是默认的左边（即从默认的右对齐切换为左对齐）；
'#'     切换格式：
       八进制数前加0（%#o），十六进制数前加0x（%#x）或0X（%#X），指针去掉前面
      对%q（%#q），如果strconv.CanBackquote返回真会输出反引号括起来的未转义
      对%U（%#U），输出Unicode格式后，如字符可打印，还会输出空格和单引号括起来
       对字符串采用%x或%X时（% x或% X）会给各打印的字节之间加空格；
'0'     使用0而不是空格填充，对于数值类型会把填充的0放在正负号后面；
```

verb会忽略不支持的flag。例如，因为没有十进制切换模式，所以%#d和%d的输出是相同的。

对每一个类似Printf的函数，都有对应的Print型函数，该函数不接受格式字符串，就效果上等价于对每一个参数都是用verb %v。另一个变体Println型函数会在各个操作数的输出之间加空格并在最后换行。

不管verb如何，如果操作数是一个接口值，那么会使用接口内部保管的值，而不是接口，因此：

```
var i interface{} = 23
fmt.Printf("%v\n", i)
```

会输出23。

除了verb %T和%p之外；对实现了特定接口的操作数会考虑采用特殊的格式化技巧。按应用优先级如下：

1. 如果操作数实现了Formatter接口，会调用该接口的方法。Formatter提供了格式化的控制。
2. 如果verb %v配合flag #使用（%#v），且操作数实现了GoStringer接口，会调用该接口。

如果操作数满足如下两条任一条，对于%s、%q、%v、%x、%X五个verb，将考虑：

1. 如果操作数实现了error接口，Error方法会用来生成字符串，随后将按给出的flag（如果有）和verb格式化。

2. 如果操作数具有String方法，这个方法将被用来生成字符串，然后将按给出的flag（如果有）和verb格式化。

复合类型的操作数，如切片和结构体，格式化动作verb递归地应用于其每一个成员，而不是作为整体一个操作数使用。因此%q会将[]string的每一个成员括起来，%6.2f会控制浮点数组的每一个元素的格式化。

为了避免可能出现的无穷递归，如：

```
type X string
func (x X) String() string { return Sprintf("<%s>", x) }
```

应在递归之前转换值的类型：

```
func (x X) String() string { return Sprintf("<%s>", string(x)) }
```

显式指定参数索引：

在Printf、Sprintf、Fprintf三个函数中，默认的行为是对每一个格式化verb依次对应调用时成功传递进来的参数。但是，紧跟在verb之前的[n]符号表示应格式化第n个参数（索引从1开始）。同样的在'*'之前的[n]符号表示采用第n个参数的值作为宽度或精度。在处理完方括号表达式[n]后，除非另有指示，会接着处理参数n+1，n+2……（就是说移动了当前处理位置）。例如：

```
fmt.Sprintf("%[2]d %[1]d\n", 11, 22)
```

会生成"22 11"，而：

```
fmt.Sprintf("%[3]*.[2]*[1]f", 12.0, 2, 6),
```

等价于：

```
fmt.Sprintf("%6.2f", 12.0),
```

会生成" 12.00"。因为显式的索引会影响随后的verb，这种符号可以通过重设索引用于多次打印同一个值：

```
fmt.Sprintf("%d %d %#[1]x %#x", 16, 17)
```

会生成"16 17 0x10 0x11"

格式化错误：

如果给某个verb提供了非法的参数，如给%d提供了一个字符串，生成的字符串会包含该问题的描述，如下所例：

```
错误的类型或未知的verb：%!verb(type=value)
	Printf("%d", hi):          %!d(string=hi)
太多参数（采用索引时会失效）：%!(EXTRA type=value)
	Printf("hi", "guys"):      hi%!(EXTRA string=guys)
太少参数: %!verb(MISSING)
	Printf("hi%d"):            hi %!d(MISSING)
宽度/精度不是整数值：%!(BADWIDTH) or %!(BADPREC)
	Printf("%*s", 4.5, "hi"):  %!(BADWIDTH)hi
	Printf("%.*s", 4.5, "hi"): %!(BADPREC)hi
没有索引指向的参数：%!(BADINDEX)
	Printf("%*[2]d", 7):       %!d(BADINDEX)
	Printf("%.[2]d", 7):       %!d(BADINDEX)
```

所有的错误都以字符串"%!"开始，有时会后跟单个字符（verb标识符），并以加小括弧的描述结束。

如果被print系列函数调用时，Error或String方法触发了panic，fmt包会根据panic重建错误信息，用一个字符串说明该panic经过了fmt包。例如，一个String方法调用了panic("bad")，生成的格式化信息差不多是这样的：

```
%!s(PANIC=bad)
```

%!s指示表示错误（panic）出现时的使用的verb。

Scanning

一系列类似的函数可以扫描格式化文本以生成值。

Scan、Scanf和Scanln从标准输入os.Stdin读取文本；Fscan、Fscanf、Fscanln从指定的io.Reader接口读取文本；Sscan、Sscanf、Sscanln从一个参数字符串读取文本。

Scanln、Fscanln、Sscanln会在读取到换行时停止，并要求一次提供一行所有条目；Scanf、Fscanf、Sscanf只有在格式化文本末端有换行时会读取到换行为止；其他函数会将换行视为空白。

Scanf、Fscanf、Sscanf会根据格式字符串解析参数，类似Printf。例如%x会读取一个十六进制的整数，%v会按对应值的默认格式读取。格式规则类似Printf，有如下区别：

```
%p 未实现
%T 未实现
%e %E %f %F %g %G 效果相同，用于读取浮点数或复数类型
%s %v 用在字符串时会读取空白分隔的一个片段
flag '#'和'+' 未实现
```

在无格式化verb或verb %v下扫描整数时会接受常用的进制设置前缀0（八进制）和0x（十六进制）。

宽度会在输入文本中被使用（%5s表示最多读取5个rune来生成一个字符串），但没有使用精度的语法（没有%5.2f，只有%5f）。

当使用格式字符串进行扫描时，多个连续的空白字符（除了换行符）在输出和输出中都被等价于一个空白符。在此前提下，格式字符串中的文本必须匹配输入的文本；如果不匹配扫描会中止，函数的整数返回值说明已经扫描并填写的参数个数。

在所有的扫描函数里，\r\n都被视为\n。

在所有的扫描函数里，如果一个操作数实现了Scan方法（或者说，它实现了Scanner接口），将会使用该接口为该操作数扫描文本。另外，如果如果扫描到（准备填写）的参数比提供的参数个数少，会返回一个错误。

提供的所有参数必须为指针或者实现了Scanner接口。注意：Fscan等函数可能会在返回前多读取一个rune，这导致多次调用这些函数时可能会跳过部分输入。只有在输入里各值之间没有空白时，会出现问题。如果提供给Fscan等函数的io.Reader接口实现了ReadRune方法，将使用该方法读取字符。如果该io.Reader接口还实现了UnreadRune方法，将是使用该方法保存字符，这样可以使成功执行的Fscan等函数不会丢失数据。如果要给一个没有这两个方法的io.Reader接口提供这两个方法，使用bufio.NewReader。

Index

type Stringer

```
type Stringer interface {
    String() string
}
```

实现了Stringer接口的类型（即有String方法），定义了该类型值的原始显示。当采用任何接受字符的verb（%v %s %q %x %X）动作格式化一个操作数时，或者被不使用格式字符串如Print函数打印操作数时，会调用String方法来生成输出的文本。

type GoStringer

```
type GoStringer interface {
    GoString() string
}
```

实现了GoStringer接口的类型（即有GoString方法），定义了该类型值的go语法表示。当采用verb %#v格式化一个操作数时，会调用GoString方法来生成输出的文本。

type State

```
type State interface {
    // Write方法用来写入格式化的文本
    Write(b []byte) (ret int, err error)
    // Width返回宽度值，及其是否被设置
    Width() (wid int, ok bool)
    // Precision返回精度值，及其是否被设置
    Precision() (prec int, ok bool)
    // Flag报告是否设置了flag c（一个字符，如+、-、#等）
    Flag(c int) bool
}
```

State代表一个传递给自定义Formatter接口的Format方法的打印环境。它实现了io.Writer接口用来写入格式化的文本，还提供了该操作数的格式字符串指定的选项和宽度、精度信息（通过调用方法）。

type Formatter

```
type Formatter interface {
    // c为verb，f提供verb的细节信息和Write方法用于写入生成的格式化文本
    Format(f State, c rune)
}
```

实现了Formatter接口的类型可以定制自己的格式化输出。Format方法的实现内部可以调用Sprint或Fprint等函数来生成自身的输出。

type ScanState

```
type ScanState interface {
    // 从输入读取下一个rune（Unicode码值），在读取超过指定宽度时会返回EOF
    // 如果在Scanln、Fscanln或Sscanln中被调用，本方法会在返回第一个'\n'后
    ReadRune() (r rune, size int, err error)
    // UnreadRune方法让下一次调用ReadRune时返回上一次返回的rune且不移动读
    UnreadRune() error
    // SkipSpace方法跳过输入中的空白，换行被视为空白
    // 在Scanln、Fscanln或Sscanln中被调用时，换行被视为EOF
    SkipSpace()
    // 方法从输入中依次读取rune并用f测试，直到f返回假；将读取的rune组织为一个
    // 如果skipSpace参数为真，本方法会先跳过输入中的空白。
    // 如果f为nil，会使用!unicode.IsSpace(c)；就是说返回值token将为一串非
    // 换行被视为空白，在Scanln、Fscanln或Sscanln中被调用时，换行被视为EOF
    // 返回的切片指向一个共享内存，可能被下一次调用Token方法时重写；
    // 或被使用该Scanstate的另一个Scan函数重写；或者在本次调用的Scan方法返
    Token(skipSpace bool, f func(rune) bool) (token []byte, err er
    // Width返回返回宽度值，及其是否被设置。单位是unicode码值。
    Width() (wid int, ok bool)
    // 因为本接口实现了ReadRune方法，Read方法永远不应被在Scanner接口中使用。
    // 一个合法的ScanStat接口实现可能会选择让本方法总是返回错误。
    Read(buf []byte) (n int, err error)
}
```

ScanState代表一个将传递给Scanner接口的Scan方法的扫描环境。 Scan函数中，可以进行一次一个rune的扫描，或者使用Token方法获得下一个token（比如空白分隔的token）。

type Scanner

```
type Scanner interface {
    Scan(state ScanState, verb rune) error
}
```

当Scan、Scanf、Scanln或类似函数接受实现了Scanner接口的类型（其Scan方法的receiver必须是指针，该方法从输入读取该类型值的字符串表示并将结果写入receiver）作为参数时，会调用其Scan方法进行定制的扫描。

func Printf

```
func Printf(format string, a ...interface{}) (n int, err error)
```

Printf根据format参数生成格式化的字符串并写入标准输出。返回写入的字节数和遇到的任何错误。

func Fprintf

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int,
```

Fprintf根据format参数生成格式化的字符串并写入w。返回写入的字节数和遇到的任何错误。

func Sprintf

```
func Sprintf(format string, a ...interface{}) string
```

Sprintf根据format参数生成格式化的字符串并返回该字符串。

func Print

```
func Print(a ...interface{}) (n int, err error)
```

Print采用默认格式将其参数格式化并写入标准输出。如果两个相邻的参数都不是字符串，会在它们的输出之间添加空格。返回写入的字节数和遇到的任何错误。

func Fprint

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
```

Fprint采用默认格式将其参数格式化并写入w。如果两个相邻的参数都不是字符串，会在它们的输出之间添加空格。返回写入的字节数和遇到的任何错误。

func Sprint

```
func Sprint(a ...interface{}) string
```

Sprint采用默认格式将其参数格式化，串联所有输出生成并返回一个字符串。如果两个相邻的参数都不是字符串，会在它们的输出之间添加空格。

func Println

```
func Println(a ...interface{}) (n int, err error)
```

Println采用默认格式将其参数格式化并写入标准输出。总是会在相邻参数的输出之间添加空格并在输出结束后添加换行符。返回写入的字节数和遇到的任何错误。

func Fprintln

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
```

Fprintln采用默认格式将其参数格式化并写入w。总是会在相邻参数的输出之间添加空格并在输出结束后添加换行符。返回写入的字节数和遇到的任何错误。

func Sprintln

```
func Sprintln(a ...interface{}) string
```

Sprintln采用默认格式将其参数格式化，串联所有输出生成并返回一个字符串。总是会在相邻参数的输出之间添加空格并在输出结束后添加换行符。

func Errorf

```
func Errorf(format string, a ...interface{}) error
```

Errorf根据format参数生成格式化字符串并返回一个包含该字符串的错误。

func Scanf

```
func Scanf(format string, a ...interface{}) (n int, err error)
```

Scanf从标准输入扫描文本，根据format 参数指定的格式将成功读取的空白分隔的值保存进成功传递给本函数的参数。返回成功扫描的条目个数和遇到的任何错误。

func Fscanf

```
func Fscanf(r io.Reader, format string, a ...interface{}) (n int, e
```

Fscanf从r扫描文本，根据format 参数指定的格式将成功读取的空白分隔的值保存进成功传递给本函数的参数。返回成功扫描的条目个数和遇到的任何错误。

func Sscanf

```
func Sscanf(str string, format string, a ...interface{}) (n int, er
```

Sscanf从字符串str扫描文本，根据format 参数指定的格式将成功读取的空白分隔的值保存进成功传递给本函数的参数。返回成功扫描的条目个数和遇到的任何错误。

func Scan

```
func Scan(a ...interface{}) (n int, err error)
```

Scan从标准输入扫描文本，将成功读取的空白分隔的值保存进成功传递给本函数的参数。换行视为空白。返回成功扫描的条目个数和遇到的任何错误。如果读取的条目比提供的参数少，会返回一个错误报告原因。

func Fscan

```
func Fscan(r io.Reader, a ...interface{}) (n int, err error)
```

Fscan从r扫描文本，将成功读取的空白分隔的值保存进成功传递给本函数的参数。换行视为空白。返回成功扫描的条目个数和遇到的任何错误。如果读取的条目比提供的参数少，会返回一个错误报告原因。

func Sscan

```
func Sscan(str string, a ...interface{}) (n int, err error)
```

Sscan从字符串str扫描文本，将成功读取的空白分隔的值保存进成功传递给本函数的参数。换行视为空白。返回成功扫描的条目个数和遇到的任何错误。如果读取的条目比提供的参数少，会返回一个错误报告原因。

func Scanln

```
func Scanln(a ...interface{}) (n int, err error)
```

Scanln类似Scan，但会在换行时才停止扫描。最后一个条目后必须有换行或者到达结束位置。

func Fscanln

```
func Fscanln(r io.Reader, a ...interface{}) (n int, err error)
```

Fscanln类似Fscan，但会在换行时才停止扫描。最后一个条目后必须有换行或者到达结束位置。

func Sscanln

```
func Sscanln(str string, a ...interface{}) (n int, err error)
```

Sscanln类似Sscan，但会在换行时才停止扫描。最后一个条目后必须有换行或者到达结束位置。

package go

package doc

```
import "go/doc"
```

doc包从Go的AST提取源码文档。

Index

Variables

```
var IllegalPrefixes = []string{
    "copyright",
    "all rights",
    "author",
}
```

Examples

```
func Examples(files ...*ast.File) []*Example
```

Examples returns the examples found in the files, sorted by Name field. The Order fields record the order in which the examples were encountered.

Playable Examples must be in a package whose name ends in "_test". An Example is "playable" (the Play field is non-nil) in either of these circumstances:

```
- The example function is self-contained: the function references o
  identifiers from other packages (or predeclared identifiers, such
  "int") and the test file does not include a dot import.
- The entire test file is the example: the file contains exactly on
  example function, zero test or benchmark functions, and at least
  top-level function, type, variable, or constant declaration other
  than the example function.
```

func Synopsis

```
func Synopsis(s string) string
```

Synopsis returns a cleaned version of the first sentence in s. That sentence ends after the first period followed by space and not preceded by exactly one uppercase letter. The result string has no \n, \r, or \t characters and uses only single spaces between words. If s starts with any of the IllegalPrefixes, the result is the empty string.

func ToHTML

```
func ToHTML(w io.Writer, text string, words map[string]string)
```

ToHTML converts comment text to formatted HTML. The comment was prepared by DocReader, so it is known not to have leading, trailing blank lines nor to have trailing spaces at the end of lines. The comment markers have already been removed.

Each span of unindented non-blank lines is converted into a single paragraph. There is one exception to the rule: a span that consists of a single line, is followed by another paragraph span, begins with a capital letter, and contains no punctuation is formatted as a heading.

A span of indented lines is converted into a <pre> block, with the common indent prefix removed.

URLs in the comment text are converted into links; if the URL also appears in the words map, the link is taken from the map (if the corresponding map value is the empty string, the URL is not converted into a link).

Go identifiers that appear in the words map are italicized; if the corresponding map value is not the empty string, it is considered a URL and the word is converted into a link.

func ToText

```
func ToText(w io.Writer, text string, indent, preIndent string, wid
```

ToText prepares comment text for presentation in textual output. It wraps paragraphs of text to width or fewer Unicode code points and then prefixes each line with the indent. In preformatted sections (such as program text), it prefixes each non-blank line with preIndent.

type Example

```
type Example struct {
    Name        string // name of the item being exemplified
    Doc         string // example function doc string
    Code        ast.Node
    Play        *ast.File // a whole program version of the example
    Comments    []*ast.CommentGroup
    Output      string // expected output
    EmptyOutput bool   // expect empty output
    Order       int    // original source code order
}
```

An Example represents an example function found in a source files.

type Filter

```
type Filter func(string) bool
```

type Func

```
type Func struct {
    Doc  string
    Name string
    Decl *ast.FuncDecl

    // methods
    // (for functions, these fields have the respective zero value)
    Recv  string // actual   receiver "T" or "*T"
    Orig  string // original receiver "T" or "*T"
    Level int    // embedding level; 0 means not embedded
}
```

Func is the documentation for a func declaration.

type Mode

```
type Mode int
```

Mode values control the operation of New.

```
const (
    // extract documentation for all package-level declarations,
    // not just exported ones
    AllDecls Mode = 1 << iota

    // show all embedded methods, not just the ones of
    // invisible (unexported) anonymous fields
    AllMethods
)
```

type Note

```
type Note struct {
    Pos, End token.Pos // position range of the comment containing
    UID      string    // uid found with the marker
    Body     string    // note body text
}
```

A Note represents a marked comment starting with "MARKER(uid): note body". Any note with a marker of 2 or more upper case [A-Z] letters and a uid of at least one character is recognized. The ":" following the uid is optional. Notes are

collected in the Package.Notes map indexed by the notes marker.

type Package

```
type Package struct {
    Doc        string
    Name       string
    ImportPath string
    Imports    []string
    Filenames  []string
    Notes      map[string][]*Note
    // DEPRECATED. For backward compatibility Bugs is still populat
    // but all new code should use Notes instead.
    Bugs []string

    // declarations
    Consts []*Value
    Types  []*Type
    Vars   []*Value
    Funcs  []*Func
}
```

Package is the documentation for an entire package.

func New

```
func New(pkg *ast.Package, importPath string, mode Mode) *Package
```

New computes the package documentation for the given package AST. New takes ownership of the AST pkg and may edit or overwrite it.

func (*Package) Filter

```
func (p *Package) Filter(f Filter)
```

Filter eliminates documentation for names that don't pass through the filter f. TODO(gri): Recognize "Type.Method" as a name.

type Type

```
type Type struct {
    Doc  string
    Name string
    Decl *ast.GenDecl

    // associated declarations
    Consts  []*Value // sorted list of constants of (mostly) this t
    Vars    []*Value // sorted list of variables of (mostly) this t
    Funcs   []*Func  // sorted list of functions returning this typ
    Methods []*Func  // sorted list of methods (including embedded 
}
```

Type is the documentation for a type declaration.

type Value

```
type Value struct {
    Doc   string
    Names []string // var or const names in declaration order
    Decl  *ast.GenDecl
    // contains filtered or unexported fields
}
```

Value is the documentation for a (possibly grouped) var or const declaration.

package format

```
import "go/format"
```

Package format implements standard formatting of Go source.。

Index

func Node

```
func Node(dst io.Writer, fset *token.FileSet, node interface{}) err
```

Node formats node in canonical gofmt style and writes the result to dst.

The node type must be *ast.File, *printer.CommentedNode, []ast.Decl, []ast.Stmt, or assignment-compatible to ast.Expr, ast.Decl, ast.Spec, or ast.Stmt. Node does not modify node. Imports are not sorted for nodes representing partial source files (i.e., if the node is not an *ast.File or a *printer.CommentedNode not wrapping an *ast.File).

The function may return early (before the entire result is written) and return a formatting error, for instance due to an incorrect AST.

func Source

```
func Source(src []byte) ([]byte, error)
```

Source formats src in canonical gofmt style and returns the result or an (I/O or syntax) error. src is expected to be a syntactically correct Go source file, or a list of Go declarations or statements.

If src is a partial source file, the leading and trailing space of src is applied to the result (such that it has the same leading and trailing space as src), and the result is indented by the same amount as the first line of src containing code. Imports are not sorted for partial source files.

package parser

```
import "go/parser"
```

Package parser implements a parser for Go source files. Input may be provided in a variety of forms (see the various Parse *functions); the output is an abstract syntax tree (AST) representing the Go source. The parser is invoked through one of the Parse* functions.

Index

func ParseDir

```
func ParseDir(fset *token.FileSet, path string, filter func(os.File
```

ParseDir calls ParseFile for all files with names ending in ".go" in the directory specified by path and returns a map of package name -> package AST with all the packages found.

If filter != nil, only the files with os.FileInfo entries passing through the filter (and ending in ".go") are considered. The mode bits are passed to ParseFile unchanged. Position information is recorded in fset.

If the directory couldn't be read, a nil map and the respective error are returned. If a parse error occurred, a non-nil but incomplete map and the first error encountered are returned.

func ParseExpr

```
func ParseExpr(x string) (ast.Expr, error)
```

ParseExpr is a convenience function for obtaining the AST of an expression x. The position information recorded in the AST is undefined. The filename used in error messages is the empty string.

func ParseFile

```
func ParseFile(fset *token.FileSet, filename string, src interface
```

ParseFile parses the source code of a single Go source file and returns the corresponding ast.File node. The source code may be provided via the filename of the source file, or via the src parameter.

If src != nil, ParseFile parses the source from src and the filename is only used when recording position information. The type of the argument for the src parameter must be string, []byte, or io.Reader. If src == nil, ParseFile parses the file specified by filename.

The mode parameter controls the amount of source text parsed and other optional parser functionality. Position information is recorded in the file set fset.

If the source couldn't be read, the returned AST is nil and the error indicates the specific failure. If the source was read but syntax errors were found, the result is a partial AST (with ast.Bad* nodes representing the fragments of erroneous source code). Multiple errors are returned via a scanner.ErrorList which is sorted by file position.

Code:

```
fset := token.NewFileSet() // positions are relative to fset

// Parse the file containing this very example
// but stop after processing the imports.
f, err := parser.ParseFile(fset, "example_test.go", nil, parser.Imp
if err != nil {
    fmt.Println(err)
    return
}

// Print the imports from the file's AST.
for _, s := range f.Imports {
    fmt.Println(s.Path.Value)
}
```

Output:

```
"fmt"
"go/parser"
"go/token"
```

type Mode

```
type Mode uint
```

A Mode value is a set of flags (or 0). They control the amount of source code parsed and other optional parser functionality.

```
const (
    PackageClauseOnly Mode             = 1 << iota // stop parsing
    ImportsOnly                                    // stop parsing
    ParseComments                                  // parse comment
    Trace                                          // print a trace
    DeclarationErrors                              // report declar
    SpuriousErrors                                 // same as AllEr
    AllErrors         = SpuriousErrors             // report all er
)
```

package printer

```
import "go/printer"
```

Package printer implements printing of AST nodes..

Index

func Fprint

```
func Fprint(output io.Writer, fset *token.FileSet, node interface{]
```

Fprint "pretty-prints" an AST node to output. It calls Config.Fprint with default settings.

```
// Parse source file and extract the AST without comments for
// this function, with position information referring to the
// file set fset.
funcAST, fset := parseFunc("example_test.go", "ExampleFprint")

// Print the function body into buffer buf.
// The file set is provided to the printer so that it knows
// about the original source formatting and can add additional
// line breaks where they were present in the source.
var buf bytes.Buffer
printer.Fprint(&buf, fset, funcAST.Body)

// Remove braces {} enclosing the function body, unindent,
// and trim leading and trailing white space.
s := buf.String()
s = s[1 : len(s)-1]
s = strings.TrimSpace(strings.Replace(s, "\n\t", "\n", -1))

// Print the cleaned-up body text to stdout.
fmt.Println(s)
```

Output:

```
funcAST, fset := parseFunc("example_test.go", "ExampleFprint")

var buf bytes.Buffer
printer.Fprint(&buf, fset, funcAST.Body)

s := buf.String()
s = s[1 : len(s)-1]
s = strings.TrimSpace(strings.Replace(s, "\n\t", "\n", -1))

fmt.Println(s)
```

type CommentedNode

```
type CommentedNode struct {
    Node     interface{} // *ast.File, or ast.Expr, ast.Decl, ast.S
    Comments []*ast.CommentGroup
}
```

A CommentedNode bundles an AST node and corresponding comments. It may be provided as argument to any of the Fprint functions.

type Config

```
type Config struct {
    Mode     Mode // default: 0
    Tabwidth int  // default: 8
    Indent   int  // default: 0 (all code is indented at least by t
}
```

A Config node controls the output of Fprint.

func (*Config) Fprint

```
func (cfg *Config) Fprint(output io.Writer, fset *token.FileSet, no
```

Fprint "pretty-prints" an AST node to output for a given configuration cfg. Position information is interpreted relative to the file set fset. The node type must be *ast.File, *CommentedNode, []ast.Decl, []ast.Stmt, or assignment-compatible to ast.Expr, ast.Decl, ast.Spec, or ast.Stmt.

type Mode

```
type Mode uint
```

A Mode value is a set of flags (or 0). They control printing.

```
const (
    RawFormat Mode = 1 << iota // do not use a tabwriter; if set, u
    TabIndent                  // use tabs for indentation independ
    UseSpaces                  // use spaces instead of tabs for al
    SourcePos                  // emit //line comments to preserve
)
```

package hash

package adler32

```
import "hash/adler32"
```

adler32包实现了Adler-32校验和算法，参见RFC 1950：

```
Adler-32由两个每字节累积的和组成：
s1是所有字节的累积，s2是所有s1的累积。两个累积值都取65521的余数。s1初始为1，s
Afler-32校验和保存为s2*65536 + s1。（最高有效字节在前/大端在前）
```

Index

Constants

```
const Size = 4
```

Adler-32校验和的字节数。

func Checksum

```
func Checksum(data []byte) uint32
```

返回数据data的Adler-32校验和。

func New

```
func New() hash.Hash32
```

返回一个计算Adler-32校验和的hash.Hash32接口。

package crc32

```
import "hash/crc32"
```

crc32包实现了32位循环冗余校验（CRC-32）的校验和算法，参见：

http://en.wikipedia.org/wiki/Cyclic_redundancy_check

Index

Constants

```
const (
    // 最常用的CRC-32多项式；用于以太网、v.42、fddi、gzip、zip、png、mpeg
    IEEE = 0xedb88320
    // 卡斯塔尼奥利多项式，用在iSCSI；有比IEEE更好的错误探测特性
    // http://dx.doi.org/10.1109/26.231911
    Castagnoli = 0x82f63b78
    // 库普曼多项式；错误探测特性也比IEEE好
    // http://dx.doi.org/10.1109/DSN.2002.1028931
    Koopman = 0xeb31d82e
)
```

预定义的多项式。

```
const Size = 4
```

CRC-32校验和的字节长度。

Variables

```
var IEEETable = MakeTable(IEEE)
```

IEEETable是IEEE多项式对应的Table。

type Table

```
type Table [256]uint32
```

长度256的uint32切片，代表一个用于高效运作的多项式。

func MakeTable

```
func MakeTable(poly uint32) *Table
```

返回一个代表poly指定的多项式的Table。

func Checksum

```
func Checksum(data []byte, tab *Table) uint32
```

返回数据data使用tab代表的多项式计算出的CRC-32校验和。

func ChecksumIEEE

```
func ChecksumIEEE(data []byte) uint32
```

返回数据data使用IEEE多项式计算出的CRC-32校验和。

func Update

```
func Update(crc uint32, tab *Table, p []byte) uint32
```

返回将切片p的数据采用tab表示的多项式添加到crc之后计算出的新校验和。

func New

```
func New(tab *Table) hash.Hash32
```

创建一个使用tab代表的多项式计算CRC-32校验和的hash.Hash32接口。

func NewIEEE

```
func NewIEEE() hash.Hash32
```

创建一个使用IEEE多项式计算CRC-32校验和的hash.Hash32接口。

package crc64

`import "hash/crc64"`

Package crc64 implements the 64-bit cyclic redundancy check, or CRC-64, checksum. See http://en.wikipedia.org/wiki/Cyclic_redundancy_check for information.

Index

Constants

```
const (
    // ISO 3309定义的ISO多项式，用于HDLC
    ISO = 0xD800000000000000
    // ECMA 182定义的ECMA多项式
    ECMA = 0xC96C5795D7870F42
)
```

预定义的多项式。

```
const Size = 8
```

CRC-64校验和的字节数。

type Table

```
type Table [256]uint64
```

长度256的uint64切片，代表一个用于高效运作的多项式。

func MakeTable

```
func MakeTable(poly uint64) *Table
```

返回一个代表poly指定的多项式的*Table。

func Checksum

```
func Checksum(data []byte, tab *Table) uint64
```

返回数据data使用tab代表的多项式计算出的CRC-64校验和。

func Update

```
func Update(crc uint64, tab *Table, p []byte) uint64
```

返回将切片p的数据采用tab表示的多项式添加到crc之后计算出的新校验和。

func New

```
func New(tab *Table) hash.Hash64
```

创建一个使用tab代表的多项式计算CRC-64校验和的hash.Hash64接口。

package fnv

`import "hash/fnv"`

fnv包实现了FNV-1和FNV-1a（非加密hash函数），算法参见：

http://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function

Index

func New32

```
func New32() hash.Hash32
```

返回一个新的32位FNV-1的hash.Hash32接口

func New32a

```
func New32a() hash.Hash32
```

返回一个新的32位FNV-1a的hash.Hash32接口

func New64

```
func New64() hash.Hash64
```

返回一个新的64位FNV-1的hash.Hash64接口

func New64a

```
func New64a() hash.Hash64
```

返回一个新的64位FNV-1a的hash.Hash64接口

package html

```
import "html"
```

html包提供了用于转义和解转义HTML文本的函数。

Index

func EscapeString

```
func EscapeString(s string) string
```

EscapeString函数将特定的一些字符转为逸码后的字符实体，如"<"变成"<"。

它只会修改五个字符：<、>、&、'、"。

UnescapeString(EscapeString(s)) == s总是成立，但是两个函数顺序反过来则不一定成立。

func UnescapeString

```
func UnescapeString(s string) string
```

UnescapeString函数将逸码的字符实体如"<"修改为原字符"<"。它会解码一个很大范围内的字符实体，远比函数EscapeString转码范围大得多。例如"á"解码为"á"，"á"和"&xE1;"也会解码为该字符。

package template

```
import "html/template"
```

template包（html/template）实现了数据驱动的模板，用于生成可对抗代码注入的安全HTML输出。本包提供了和text/template包相同的接口，无论何时当输出是HTML的时候都应使用本包。

此处的文档关注本包的安全特性。至于如何使用模板，请参照text/template包。

Introduction

本包是对text/template包的包装，两个包提供的模板API几无差别，可以安全的随意替换两包。

```
tmpl, err := template.New("name").Parse(...)
// 省略错误检测
err = tmpl.Execute(out, data)
```

如果成功创建了tmpl，tmpl现在是注入安全的了。否则err将返回ErrorCode里定义的某个错误。即使成功生成了模板，执行时仍可能导致ErrorCode里定义的错误。

HTML模板将数据视为明文文本，必须经过编码以便安全的嵌入HTML文档。转义操作会参考上下文，因此action可以出现在JavaScript、CSS、URI上下文环境里。

本包使用的安全模型假设模板的作者是可信任的，但用于执行的数据不可信。更多细节参见下面。

示例：

```
import "text/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{e
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwn
```

生成：

```
Hello, <script>alert('you have been pwned')</script>!
```

但在html/template包里会根据上下文自动转义：

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{e
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwn
```

生成安全的转义后HTML输出：

```
Hello, &lt;script&gt;alert(&#39;you have been pwned&#39;)&lt;/scrip
```

Contexts

本包可以理解HTML、CSS、JavaScript和URI。它会给每一个简单的action pipeline都添加处理函数，如下例：

```
<a href="/search?q={{.}}">{{.}}</a>
```

在解析时每个{{.}}都会在必要时重写添加转义函数，此例中会修改为：

```
<a href="/search?q={{. | urlquery}}">{{. | html}}</a>
```

Errors

细节请参见ErrorCode类型的文档。

A fuller picture

本包剩余部分的注释第一次阅读时可以跳过；这些部分包括理解转码文本和错误信息的必要细节。多数使用者无需理解这些细节。

Contexts

假设{{.}}是 `O'Reilly: How are <i>you</i>?` ，下表展示了{{.}}用于左侧模板时的输出：

```
Context                          {{.}} After
{{.}}                            O'Reilly: How are &lt;i&gt;you&lt;
<a title='{{.}}'>                O&#39;Reilly: How are you?
<a href="/{{.}}">                O&#39;Reilly: How are %3ci%3eyou%3
<a href="?q={{.}}">              O&#39;Reilly%3a%20How%20are%3ci%3e
<a onx='f("{{.}}")'>             O\x27Reilly: How are \x3ci\x3eyou
<a onx='f({{.}})'>               "O\x27Reilly: How are \x3ci\x3eyou
<a onx='pattern = /{{.}}/;'>     O\x27Reilly: How are \x3ci\x3eyou
```

如果用在不安全的上下文里，值就可能被过滤掉：

```
Context                          {{.}} After
<a href="{{.}}">                 #ZgotmplZ
```

因为"O'Reilly:"不是一个可以接受的协议名，如"http:"。

如果{{.}}是一个无害的词汇，如 `left` ，那么它就可以出现在更多地方。

```
Context                              {{.}} After
{{.}}                                left
<a title='{{.}}'>                    left
<a href='{{.}}'>                     left
<a href='/{{.}}'>                    left
<a href='?dir={{.}}'>                left
<a style="border-{{.}}: 4px">        left
<a style="align: {{.}}">             left
<a style="background: '{{.}}'>       left
<a style="background: url('{{.}}')>  left
<style>p.{{.}} {color:red}</style>   left
```

如果{{.}}是非字符串类型的值，可以用于JavaScript上下文环境里：

```
struct{A,B string}{ "foo", "bar" }
```

将该值应用在在转义后的模板里：

```
<script>var pair = {{.}};</script>
```

模板输出为：

```
<script>var pair = {"A": "foo", "B": "bar"};</script>
```

请参见json包来理解非字符串内容是如何序列化并嵌入JavaScript里的。

Typed Strings

本包默认所有的pipeline都生成明文字符串，它会在必要时添加转义pipeline阶段以安全并正确的将明文字符串嵌入输出的文本里。

当用于执行的数据不是明文字符串时，你可以通过显式改变数据的类型以避免其被错误的转义。

类型HTML、JS、URL和其他content.go里定义的类型可以保持不被转义的安全内容。

模板：

```
Hello, {{.}}!
```

可以采用如下调用：

```
tmpl.Execute(out, HTML(`<b>World</b>`))
```

来输出：

```
Hello, <b>World</b>!
```

而不是：

```
Hello, &lt;b&gt;World&lt;b&gt;!
```

如果{{.}}是一个内建类型字符串就会产生该输出。

Security Model

本包里安全的定义参加如下网页：

http://js-quasis-libraries-and-repl.googlecode.com/svn/trunk/safetemplate.html#problem_definition

本包假设模板作者可信而执行数据不可信，目标是在保证安全性的前提下保证效率：

结构保留特性："......当模板作者用安全的模板语言写了一个HTML标签时，不管数据的值为何浏览器都会将输出的相应部分解释为标签，该情况在其他结构里也成立，如属性边界以及JS和CSS边界。"

代码影响特性："……只有模板作者指定的代码能作为注入模板输出到页面的结果执行，所有模板作者指定的代码都应如此。"

最少惊讶特性："一个熟悉HTML、CSS、JS的开发者（或代码阅读者），应可以正确的推断出{{.}}会如何转义。"

Index

type ErrorCode

```
type ErrorCode int
```

ErrorCode是代表错误种类的错误码。

```
const (
    // OK表示没有出错
    OK  ErrorCode = iota
    // 当上下文环境有歧义时导致ErrAmbigContext：
    // 举例：
    //   <a href="{{if .C}}/path/{{else}}/search?q={{end}}{{.X}}"&r
    // 说明：
    //   {{.X}}的URL上下文环境有歧义，因为根据{{.C}}的值，
    //   它可以是URL的后缀，或者是查询的参数。
    //   将{{.X}}移动到如下情况可以消除歧义：
    //   <a href="{{if .C}}/path/{{.X}}{{else}}/search?q={{.X}}{{er
    ErrAmbigContext
    // 期望空白、属性名、标签结束标志而没有时，标签名或无引号标签值包含非法字符
    // 会导致ErrBadHTML；举例：
    //   <a href = /search?q=foo&rt;
    //   <href=foo&rt;
    //   <form na<e=...&rt;
    //   <option selected<
    // 讨论：
    //   一般是因为HTML元素输入了错误的标签名、属性名或者未用引号的属性值，导
    //   将所有的属性都用引号括起来是最好的策略
    ErrBadHTML
    // {{if}}等分支不在相同上下文开始和结束时，导致ErrBranchEnd
    // 示例：
    //   {{if .C}}<a href="{{end}}{{.X}}
    // 讨论：
    //   html/template包会静态的检验{{if}}、{{range}}或{{with}}的每一个
    //   以对后续的pipeline进行转义。该例出现了歧义，{{.X}}可能是HTML文本节
    //   或者是HTML属性值的URL的前缀，{{.X}}的上下文环境可以确定如何转义，但
    //   上下文环境却是由运行时{{.C}}的值决定的，不能在编译期获知。
    //   这种问题一般是因为缺少引号或者角括号引起的，另一些则可以通过重构将两个
    //   放进if、range、with的不同分支里来避免，如果问题出现在参数长度一定非
    //   {{range}}的分支里，可以通过添加无效{{else}}分支解决。
    ErrBranchEnd
    // 如果以非文本上下文结束，则导致ErrEndContext
    // 示例：
    //   <div
    //   <div title="no close quote&rt;
    //   <script>f()
    // 讨论：
    //   执行模板必须生成HTML的一个文档片段，以未闭合标签结束的模板都会引发本
    //   不用在HTML上下文或者生成不完整片段的模板不应直接执行。
    //   {{define "main"}} <script&rt;{{template "helper"}}</script
    //   {{define "helper"}} document.write(' <div title=" ') {{enc
```

```
    //   模板"helper"不能生成合法的文档片段，所以不直接执行，用js生成。
    ErrEndContext
    // 调用不存在的模板时导致ErrNoSuchTemplate
    // 示例：
    //   {{define "main"}}<div {{template "attrs"}}&rt;{{end}}
    //   {{define "attrs"}}href="{{.URL}}"{{end}}
    // 讨论：
    //   html/template包略过模板调用计算上下文环境。
    //   此例中，当被"main"模板调用时，"attrs"模板的{{.URL}}必须视为一个UR
    //   但如果解析"main"时，"attrs"还未被定义，就会导致本错误
    ErrNoSuchTemplate
    // 不能计算输出位置的上下文环境时，导致ErrOutputContext
    // 示例：
    //   {{define "t"}}{{if .T}}{{template "t" .T}}{{end}}{{.H}}",
    // 讨论：
    //   一个递归的模板，其起始和结束的上下文环境不同时；
    //   不能计算出可信的输出位置上下文环境时，就可能导致本错误。
    //   检查各个命名模板是否有错误；
    //   如果模板不应在命名的起始上下文环境调用，检查在不期望上下文环境中对该
    //   或者将递归模板重构为非递归模板；
    ErrOutputContext
    // 尚未支持JS正则表达式插入字符集
    // 示例：
    //     <script>var pattern = /foo[{{.Chars}}]/</script&rt;
    // 讨论：
    //   html/template不支持向JS正则表达式里插入字面值字符集
    ErrPartialCharset
    // 部分转义序列尚未支持
    // 示例：
    //   <script>alert("\{{.X}}")</script&rt;
    // 讨论：
    //   html/template包不支持紧跟在反斜杠后面的action
    //   这一般是错误的，有更好的解决方法，例如：
    //     <script>alert("{{.X}}")</script&rt;
    //   可以工作，如果{{.X}}是部分转义序列，如"xA0"，
    //   可以将整个序列标记为安全文本：JSStr(`\xA0`)
    ErrPartialEscape
    // range循环的重入口出错，导致ErrRangeLoopReentry
    // 示例：
    //   <script>var x = [{{range .}}'{{.}},{{end}}]</script&rt;
    // 讨论：
    //   如果range的迭代部分导致其结束于上一次循环的另一上下文，将不会有唯一的
    //   此例中，缺少一个引号，因此无法确定{{.}}是存在于一个JS字符串里，还是
    //   第二次迭代生成类似下面的输出：
    //     <script>var x = ['firstValue,'secondValue]</script&rt;
    ErrRangeLoopReentry
    // 斜杠可以开始一个除法或者正则表达式
    // 示例：
    //   <script&rt;
    //     {{if .C}}var x = 1{{end}}
    //     /-{{.N}}/i.test(x) ? doThis : doThat();
    //   </script&rt;
    // 讨论：
```

```
    //   上例可以生成`var x = 1/-2/i.test(s)...`，其中第一个斜杠作为除号
    //   或者它也可以生成`/-2/i.test(s)`，其中第一个斜杠生成一个正则表达式
    //   检查分支中是否缺少分号，或者使用括号来明确你的意图
    ErrSlashAmbig
)
```

我们为转义模板时的所有错误都定义了错误码，但经过转义修正的模板仍可能在运行时出错：

输出"ZgotmplZ"的例子：

```
<img src="{{.X}}">
其中{{.X}}执行结果为`javascript:...`
```

讨论：

```
"ZgotmplZ"是一个特殊值，表示运行时在CSS或URL上下文环境生成的不安全内容。本例
  <img src="#ZgotmplZ">
如果数据来源可信，请转换内容类型来避免被滤除：URL(`javascript:...`)
```

type Error

```
type Error struct {
    // ErrorCode描述错误的种类
    ErrorCode ErrorCode
    // Name是发生错误的模板的名字
    Name string
    // Line是错误位置在模板原文中的行号或者0
    Line int
    // Description是供调试者阅读的错误描述
    Description string
}
```

Error描述在模板转义时出现的错误。

func (*Error) Error

```
func (e *Error) Error() string
```

func HTMLEscape

```
func HTMLEscape(w io.Writer, b []byte)
```

函数向w中写入b的HTML转义等价表示。

func HTMLEscapeString

```
func HTMLEscapeString(s string) string
```

返回s的HTML转义等价表示字符串。

func HTMLEscaper

```
func HTMLEscaper(args ...interface{}) string
```

函数返回其所有参数文本表示的HTML转义等价表示字符串。

func JSEscape

```
func JSEscape(w io.Writer, b []byte)
```

函数向w中写入b的JavaScript转义等价表示。

func JSEscapeString

```
func JSEscapeString(s string) string
```

返回s的JavaScript转义等价表示字符串。

func JSEscaper

```
func JSEscaper(args ...interface{}) string
```

函数返回其所有参数文本表示的JavaScript转义等价表示字符串。

func URLQueryEscaper

```
func URLQueryEscaper(args ...interface{}) string
```

函数返回其所有参数文本表示的可以嵌入URL查询的转义等价表示字符串。

type FuncMap

```
type FuncMap map[string]interface{}
```

FuncMap类型定义了函数名字符串到函数的映射，每个函数都必须有1到2个返回值，如果有2个则后一个必须是error接口类型；如果有2个返回值的方法返回的error非nil，模板执行会中断并返回给调用者该错误。该类型拷贝自text/template包的同名类型，因此不需要导入该包以使用该类型。

type HTML

```
type HTML string
```

HTML用于封装一个已知安全的HTML文档片段。它不应被第三方使用，也不能用于含有未闭合的标签或注释的HTML文本。该类型适用于封装一个效果良好的HTML生成器生成的HTML文本或者本包模板的输出的文本。

type HTMLAttr

```
type HTMLAttr string
```

HTMLAttr用来封装一个来源可信的HTML属性，如 `dir="ltr"` 。

type JS

```
type JS string
```

JS用于封装一个已知安全的EcmaScript5表达式，如 `(x + y * z())` 。模板作者有责任确保封装的字符串不会破坏原有的语义，也不能包含有歧义的声明或表达式，如"{ foo: bar() }\n'foo'"，这一句既是合法的表达式也是语义完全不同的合法程

序。

type JSStr

```
type JSStr string
```

JSStr用于封装一个打算嵌入JavaScript表达式中的字符序列，该字符串必须匹配一系列StringCharacters：

```
StringCharacter :: 除了`\`和行终止符的SourceCharacter | EscapeSequenc
```

注意不允许换行，JSStr("foo\nbar")是可以的，但JSStr("foo\\nbar")不可以。

type URL

```
type URL string
```

URL用来封装一个已知安全的URL或URL子字符串（参见RFC 3986）

形如 `javascript:checkThatFormNotEditedBeforeLeavingPage()` 的来源可信的URL应写进页面里，但一般动态的 `javascript:` URL排除在外（不写进页面），因为它们是频繁使用的注入向量。

type CSS

```
type CSS string
```

CSS用于包装匹配如下任一条的已知安全的内容：

```
1\. CSS3样式表，如`p { color: purple }`
2\. CSS3规则，如`a[href=~"https:"].foo#bar`
3\. CSS3声明，如`color: red; margin: 2px`
4\. CSS3规则，如`rgba(0, 0, 255, 127)`
```

参见：http://www.w3.org/TR/css3-syntax/#parsing

以及：https://web.archive.org/web/20090211114933/http://w3.org/TR/css3-syntax#style

type Template

```
type Template struct {
    // 底层的模板解析树，会更新为HTML安全的
    Tree *parse.Tree
    // 内含隐藏或非导出字段
}
```

Template类型是text/template包的Template类型的特化版本，用于生成安全的HTML文本片段。

func Must

```
func Must(t *Template, err error) *Template
```

Must函数用于包装返回(*Template, error)的函数/方法调用，它会在err非nil时panic，一般用于变量初始化：

```
var t = template.Must(template.New("name").Parse("html"))
```

func New

```
func New(name string) *Template
```

创建一个名为name的模板。

func ParseFiles

```
func ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles函数创建一个模板并解析filenames指定的文件里的模板定义。返回的模板的名字是第一个文件的文件名（不含扩展名），内容为解析后的第一个文件的内容。至少要提供一个文件。如果发生错误，会停止解析并返回nil。

func ParseGlob

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob创建一个模板并解析匹配pattern的文件（参见glob规则）里的模板定义。返回的模板的名字是第一个匹配的文件的文件名（不含扩展名），内容为解析后的第一个文件的内容。至少要存在一个匹配的文件。如果发生错误，会停止解析并返回nil。ParseGlob等价于使用匹配pattern的文件的列表为参数调用ParseFiles。

func (*Template) Name

```
func (t *Template) Name() string
```

返回模板t的名字。

func (*Template) Delims

```
func (t *Template) Delims(left, right string) *Template
```

Delims方法用于设置action的分界字符串，应用于之后的Parse、ParseFiles、ParseGlob方法。嵌套模板定义会继承这种分界符设置。空字符串分界符表示相应的默认分界符：{{或}}。返回值就是t，以便进行链式调用。

func (*Template) Funcs

```
func (t *Template) Funcs(funcMap FuncMap) *Template
```

Funcs方法向模板t的函数字典里加入参数funcMap内的键值对。如果funcMap某个键值对的值不是函数类型或者返回值不符合要求会panic。但是，可以对t函数列表的成员进行重写。方法返回t以便进行链式调用。

func (*Template) Clone

```
func (t *Template) Clone() (*Template, error)
```

Clone方法返回模板的一个副本，包括所有相关联的模板。模板的底层表示树并未拷贝，而是拷贝了命名空间，因此拷贝调用Parse方法不会修改原模板的命名空间。Clone方法用于准备模板的公用部分，向拷贝中加入其他关联模板后再进行使用。

如果t已经执行过了，会返回错误。

func (*Template) Lookup

```
func (t *Template) Lookup(name string) *Template
```

Lookup方法返回与t关联的名为name的模板，如果没有这个模板会返回nil。

func (*Template) Templates

```
func (t *Template) Templates() []*Template
```

Templates方法返回与t相关联的模板的切片，包括t自己。

func (*Template) New

```
func (t *Template) New(name string) *Template
```

New方法创建一个和t关联的名字为name的模板并返回它。这种可以传递的关联允许一个模板使用template action调用另一个模板。

func (*Template) AddParseTree

```
func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Te
```

AddParseTree方法使用name和tree创建一个模板并使它和t相关联。

如果t已经执行过了，会返回错误。

func (*Template) Parse

```
func (t *Template) Parse(src string) (*Template, error)
```

Parse方法将字符串text解析为模板。嵌套定义的模板会关联到最顶层的t。Parse可以多次调用，但只有第一次调用可以包含空格、注释和模板定义之外的文本。如果后面的调用在解析后仍剩余文本会引发错误、返回nil且丢弃剩余文本；如果解析得到的模板已有相关联的同名模板，会覆盖掉原模板。

func (*Template) ParseFiles

```
func (t *Template) ParseFiles(filenames ...string) (*Template, erro
```

ParseGlob方法解析filenames指定的文件里的模板定义并将解析结果与t关联。如果发生错误，会停止解析并返回nil，否则返回(t, nil)。至少要提供一个文件。

func (*Template) ParseGlob

```
func (t *Template) ParseGlob(pattern string) (*Template, error)
```

ParseFiles方法解析匹配pattern的文件里的模板定义并将解析结果与t关联。如果发生错误，会停止解析并返回nil，否则返回(t, nil)。至少要存在一个匹配的文件。

func (*Template) Execute

```
func (t *Template) Execute(wr io.Writer, data interface{}) error
```

Execute方法将解析好的模板应用到data上，并将输出写入wr。如果执行时出现错误，会停止执行，但有可能已经写入wr部分数据。模板可以安全的并发执行。

func (*Template) ExecuteTemplate

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data
```

ExecuteTemplate方法类似Execute，但是使用名为name的t关联的模板产生输出。

package image

```
import "image"
```

image实现了基本的2D图片库。

基本接口叫作Image。图片的色彩定义在image/color包。

Image接口可以通过调用如NewRGBA和NewPaletted函数等获得；也可以通过调用Decode函数解码包含GIF、JPEG或PNG格式图像数据的输入流获得。解码任何具体图像类型之前都必须注册对应类型的解码函数。注册过程一般是作为包初始化的副作用，放在包的init函数里。因此，要解码PNG图像，只需在程序的main包里嵌入如下代码：

```
import _ "image/png"
```

_表示导入包但不使用包中的变量/函数/类型，只是为了包初始化函数的副作用。

参见http://golang.org/doc/articles/image_package.html

Example

```
// This example demonstrates decoding a JPEG image and examining it
package image_test
import (
    "encoding/base64"
    "fmt"
    "image"
    "log"
    "strings"
    // Package image/jpeg is not used explicitly in the code below,
    // but is imported for its initialization side-effect, which al
    // image.Decode to understand JPEG formatted images. Uncomment
    // two lines to also understand GIF and PNG images:
    // _ "image/gif"
    // _ "image/png"
    _ "image/jpeg"
)
func Example() {
    // Decode the JPEG data. If reading from file, create a reader
    //
    // reader, err := os.Open("testdata/video-001.q50.420.jpeg")
    // if err != nil {
    //     log.Fatal(err)
    // }
    // defer reader.Close()
    reader := base64.NewDecoder(base64.StdEncoding, strings.NewReac
    m, _, err := image.Decode(reader)
```

```
    if err != nil {
        log.Fatal(err)
    }
    bounds := m.Bounds()
    // Calculate a 16-bin histogram for m's red, green, blue and al
    //
    // An image's bounds do not necessarily start at (0, 0), so the
    // at bounds.Min.Y and bounds.Min.X. Looping over Y first and X
    // likely to result in better memory access patterns than X fir
    var histogram [16][4]int
    for y := bounds.Min.Y; y < bounds.Max.Y; y++ {
        for x := bounds.Min.X; x < bounds.Max.X; x++ {
            r, g, b, a := m.At(x, y).RGBA()
            // A color's RGBA method returns values in the range [0
            // Shifting by 12 reduces this to the range [0, 15].
            histogram[r>>12][0]++
            histogram[g>>12][1]++
            histogram[b>>12][2]++
            histogram[a>>12][3]++
        }
    }
    // Print the results.
    fmt.Printf("%-14s %6s %6s %6s %6s\n", "bin", "red", "green", "b
    for i, x := range histogram {
        fmt.Printf("0x%04x-0x%04x: %6d %6d %6d %6d\n", i<<12, (i+1)
    }
    // Output:
    // bin               red  green   blue  alpha
    // 0x0000-0x0fff:    353    759   7228      0
    // 0x1000-0x1fff:    629   2944   1036      0
    // 0x2000-0x2fff:   1075   2319    984      0
    // 0x3000-0x3fff:    838   2291    988      0
    // 0x4000-0x4fff:    540   1302    542      0
    // 0x5000-0x5fff:    319    971    263      0
    // 0x6000-0x6fff:    316    377    178      0
    // 0x7000-0x7fff:    581    280    216      0
    // 0x8000-0x8fff:   3457    228    274      0
    // 0x9000-0x9fff:   2294    237    334      0
    // 0xa000-0xafff:    938    283    370      0
    // 0xb000-0xbfff:    322    338    401      0
    // 0xc000-0xcfff:    229    386    295      0
    // 0xd000-0xdfff:    263    416    281      0
    // 0xe000-0xefff:    538    433    312      0
    // 0xf000-0xffff:   2758   1886   1748  15450
}
const data = `
```


```
/9j/4AAQSkZJRgABAQIAHAAcAAD/2wBDABALDA4MChAODQ4SERATGCgaGBYWGDEjJR(
SFxOQERXRTc4UG1RV19iZ2hnPk1xeXBkeFxlZ2P/2wBDARESEhgVGC8aGi9jQjhCY2N
Y2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2P/wAARCABnAJYDASI
HwAAAQUBAQEBAQEAAAAAAAAAAAECAwQFBgcICQoL/8QAtRAAAgEDAwIEAwUFBAQAAAF
MUEGE1FhByJxFDKBkaEII0KxwRVS0fAkM2JyggkKFhcYGRolJicoKSo0NTY3ODk6Q0F
V1hZWmNkZWZnaGlqc3R1dnd4eXqDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztLW
x8jJytLT1NXW19jZ2uHi4+Tl5ufo6erx8vP09fb3+Pn6/8QAHwEAAwEBAQEBAQEBAQA
```

```
BgcICQoL/8QAtREAAgECBAQDBAcFBAQAAQJ3AAECAxEEBSExBhJBUQdhcRMiMoEIFE
YnLRChYkNOEl8RcYGRomJygpKjU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2hpan
hYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2N
8vP09fb3+Pn6/9oADAMBAAIRAxEAPwDlwKMD0pwzSiuK57QzGDxS7D6in8Y5ximnAP
2OKXHvRcVxnTtS7c07HNFK4DQPakC4PNOA+tOx70XAjK/So5gBGP94fzqfvUVx/qxx
cmgjilP3jSEZqS0IO/NGDnpUiocDg/McDjvV6HTPOdVWYgsM5KcfzzQ2JySM2jp6VY
kMPUVBjjtTGtRu0Zopw+lFFxhinrGzuqqMsxAA9yaXFSRv5cqSEcIwYj6GpuZ30O30
5XGTn29BV28jt7pPLuIVljPBBFVreYx+VbqAjycgt3x14zRcNOxGyVFHQkIc/wA61e
ftEuTEw3Rk9SPT8P8Kpbea3tchbyVae4JkjbbGpGdwOM89Af6ViFTWUtGdcXoM2+wo
Jt7ZqTbRtouFyPFRXI/c9D94fzqzioLsfuD/ALw/nVReqIn8LJCOTSY+tSMOTmkIpX
wjjUAuBjJJz1+laD6Pai+WaK9SBX6puzn6ZP+NV/Dkdtc6ZNbyAFwxLAHDYPv6VoQ2
Gf0NaWTOeW7Of8QwGG4MRZnEbYXPJwRnOR0zWNXW+KrqBLUWi5EjbWCgcAA9c/gRXK
FLtHvRSNiYD2pSDTgpp6p0ywUHoTULXYxcktzrdCf7Xo8LP/AKyEmMNjJ46dfbFWJ5
YrbfYGbyrjcWG88S57g+vtV26ZIvMlumKwwjLZ6V0WfU54yTvYwtbubea2WNWbzg4b
u4y2HQxqxOD1xzxmrWAQCCGB6EGsaikndmsJxeiYzBo280/Z7UbayuaXGY5oIp+2lx
ALy/zq1t96r3y4tT/vL/ADq4P3kRP4WSleTSFKkkKoCW4GaqNcMxIjXj1pxjKT0FKr
5kTAr6455/HH510UdwPtRgWCbzF5+YYUf4Vwun39xpmoR3qASMmQUJwGU9Rnt/8AWr
gAGZwFH5ZJrpVKVlY5ZYhN6kXiu2eO/ikZlIljAAB5yM549OawSOOlPuLqe+umuLqT
hl/cRsTuJHPv7mlKi3sVTxNtGP20VJhThgSQaK52mnZnUqsWrpkyeUrr5pABOAPU1A
P8qL7BiKnsMg46H3qrbzupbj5mPTPTpXVSglG551SpzSXJ4/MBUgYIxyKpySyGBYJ
bSeNJ4xchni3DeqnBI+td7F4b0mKIRjT45VbktJlzk455+n6VtYzv2PNwFZWBHGKV
NttLNkba1jgWVWDmM8bhg4/nzXLSSbXVj6fyNKUdNRp21RtIRJGrjuM0u3FQ2Dbodv
k0ejCXNFMj2/jQV9qkxSYNRcsZiq2oI32N2CkhWXJxwOe9XMcVt6hoPn6dFaW0wgRpN
ryv2Jm9LHJai+ZRGCBjnr71ErdAxAY9B611t1Y2cunbbaOQ3FvKZI3UqGlZMbiWwfcf
lt5Uq52TuZG+hGMA12xXJGxxzjzybOQtNOvb5j9ktZJhnBIHyg+5PFX38JayqK/2eL
itrSHFpGsUbndhRgc+g7VNIyfZJAoJZUbb3I46CtFJMylBo8sdWhmYMuCnylc9wef5V
sUZO5RtIPUH3pkBDOxxxmqM9TQtn+WilhHfHaik43KTG3Z4IyPyrNVjGCsZ+dmwv6V3
JJIwxChdoJGcYx/Wkg8DafA4knvLiQr/ALqj+VQpKw3FtnFFfvbiSMgzPbAf/jXp2n3d
oOxPU/8A68VVtbbRtMVntbePKDLTSHJH/Aj/AEqHTvE66rq72VugMMcbMbSGTnL4wMAf
baxaJBdzN+7bcrxkAhun0rz3VNCv7e7lgigknWI43xxsyN6HOMYPqfkpPGVYsBkghT
cxLkknp/ShczQ7xtY8vtEmhkj8yGRBuCnehUcnHcVtmwfJ/fQ8e7f/E12txZW91C0U
Bo1gM/uW55/1jf41nOipu7LhV5FZHIGzI6zwj/vr/Ck+yr3uYf8Ax7/CutbQdMb71tr
X/PoP++2/wAan6rAr6wzkWt0II+1Rc/7Lf4Vd1eeCSKBbdPdZDdShYoiZNoyfY10P/AAj
oPh/SjKspsozIuNrZORjp3qo0FHYPb3Zt7ae3SzjuItsiRSAgnccl/UA+3Q1yNjKLF
WwO/+e1evPp9nI257aJm6aJm6bioz1z1+tY+s6Hplnot9PbWMMcqwSwOFcLyOO1bJWMZSTOP
9lCj02g9P/1e9a3hzxAbl2ikZRcdQueHHt7B7HNJxTFWU5BHBB7HNJxTl
eLy5zwZI/lb8fX8azIvBUUTHdfSFP4QsYB4x7H9K6yyv
N0oy1RzOtaN/Y1tHNFO06u+zZ2GES5b6n/62PzorKVdp2Lj
jSgyxfJt6EgdDzWTdeLIZGO7zHI/hVajGmWWP+PWL8qwlAIURrhpMAHHJA71pRcZrTd
GHk245CZ6/X1qPTLq40q+W5t5WVLwpDAKEEc55/zilk5k2r91eKhLDzWz2rpsczbub
MiysmQMZAAwa3a5j4ftu0ByP+fh/5CulkLLG7INzkOHqe1Fh3uoOqn9qQQxyhndmHI
KoiBZOV9JBnt707MVy5RWdNdy7wRGf3bfMinnO1jg+vY03WXLaJO3mhQ20bfAvpYf0d
VwumgC+YiQyeVtZH567hzj8aSL949oGhE/2v5pJCDkksQwBHC4/+vXQ8LZ2uYxxCav
b+VP0bnSrb94ZMJgOecj1rl/GfidUE2k2gy5+SeQjJp7Kj3rlas2jdao48qrjLAGkSk
lIJnU8OfxPWo5inBokmtQTmM4OOh71b0q6vbFmWCbaUWCbaxHyqQGAP0PT8KhSTzVyo5ocSk
pl99XjPzThzK3zOeOSeveirNmkgg/fIpYsTkYORRxkOkCCzdAK5
UVtgtmY4nZEa8Ak9aqFv3rfSiiu1nMfDv/AJF+T/r4f+Qrp6KK5AxuztNHZbpvcK
ubeK6t3gnXdG4ww9RUy4vzzuyfX9OfBpLP1Nk/rL7/tDp4lCDnYHcVGSrmBJ1SXqMEb
3uyVGNthuq3Eei6DK8H7sRR7YWYmQrjIrm8WWy+p70UVnLY0iENaXhjYEQ7HX7/tzkQO/5n
0HwyBXGeRjmrcUhMg2ghezezd//rUVcTKKPQKKe8PbPbj8KP/9k=
`
```

Index

Examples

Variables

```
var (
    // Black是一个完全不透明的面积无限大的黑色图像
    Black = NewUniform(color.Black)
    // White是一个完全不透明的面积无限大的白色图像
    White = NewUniform(color.White)
    // Transparent是一个完全透明的面积无限大的图像
    Transparent = NewUniform(color.Transparent)
    // Opaque是一个完全不透明的面积无限大的图像
    Opaque = NewUniform(color.Opaque)
)
```

```
var ErrFormat = errors.New("image: unknown format")
```

ErrFormat说明解码时遇到了未知的格式。

type Image

```
type Image interface {
    // ColorModel方法返回图像的色彩模型
    ColorModel() color.Model
    // Bounds方法返回图像的范围，范围不一定包括点(0, 0)
    Bounds() Rectangle
    // At方法返回(x, y)位置的色彩
    // At(Bounds().Min.X, Bounds().Min.Y)返回网格左上角像素的色彩
    // At(Bounds().Max.X-1, Bounds().Max.Y-1) 返回网格右下角像素的色彩
    At(x, y int) color.Color
}
```

Image接口表示一个采用某色彩模型的颜色构成的有限矩形网格（即一幅图像）。

func Decode

```
func Decode(r io.Reader) (Image, string, error)
```

DecodeConfig函数解码并返回一个采用某种已注册格式编码的图像。字符串返回值是该格式注册时的名字。格式一般是在该编码格式的包的init函数中注册的。

type PalettedImage

```
type PalettedImage interface {
    // ColorIndexAt方法返回(x, y)位置的像素的（调色板）索引
    ColorIndexAt(x, y int) uint8
    Image
}
```

PalettedImage接口代表一幅图像，它的像素可能来自一个有限的调色板。

如果有对象m满足PalettedImage接口，且m.ColorModel()返回的color.Model接口底层为一个Palette类型值（记为p），则m.At(x, y)返回值应等于p[m.ColorIndexAt(x, y)]。如果m的色彩模型不是Palette，则ColorIndexAt的行为是不确定的。

type Config

```
type Config struct {
    ColorModel    color.Model
    Width, Height int
}
```

Config保管图像的色彩模型和尺寸信息。

func DecodeConfig

```
func DecodeConfig(r io.Reader) (Config, string, error)
```

DecodeConfig函数解码并返回一个采用某种已注册格式编码的图像的色彩模型和尺寸。字符串返回值是该格式注册时的名字。格式一般是在该编码格式的包的init函数中注册的。

func RegisterFormat

```
func RegisterFormat(name, magic string, decode func(io.Reader) (Ima
```

RegisterFormat注册一个供Decode函数使用的图片格式。name是格式的名字，如"jpeg"或"png"；magic是该格式编码的魔术前缀，该字符串可以包含"?"通配符，每个通配符匹配一个字节；decode函数用于解码图片；decodeConfig函数只解码图片的配置。

type Point

```
type Point struct {
    X, Y int
}
```

Point是X, Y坐标对。坐标轴是向右（X）向下（Y）的。既可以表示点，也可以表示向量。

```
var ZP Point
```

ZP是原点。

func Pt

```
func Pt(X, Y int) Point
```

返回Point{X , Y}

func (Point) Eq

```
func (p Point) Eq(q Point) bool
```

报告p和q是否相同。

func (Point) Add

```
func (p Point) Add(q Point) Point
```

返回点Point{p.X+q.X, p.Y+q.Y}

func (Point) Sub

```
func (p Point) Sub(q Point) Point
```

返回点Point{p.X-q.X, p.Y-q.Y}

func (Point) Mul

```
func (p Point) Mul(k int) Point
```

返回点Point{p.X*k, p.Y*k}

func (Point) Div

```
func (p Point) Div(k int) Point
```

返回点Point{p.X/k, p.Y/k }

func (Point) In

```
func (p Point) In(r Rectangle) bool
```

报告p是否在r范围内。

func (Point) Mod

```
func (p Point) Mod(r Rectangle) Point
```

返回r范围内的某点q，满足p.X-q.X是r宽度的倍数，p.Y-q.Y是r高度的倍数。

func (Point) String

```
func (p Point) String() string
```

返回p的字符串表示。格式为"(3,4)"

type Rectangle

```
type Rectangle struct {
    Min, Max Point
}
```

Rectangle代表一个矩形。该矩形包含所有满足Min.X <= X < Max.X且Min.Y <= Y < Max.Y的点。如果两个字段满足Min.X <= Max.X且Min.Y <= Max.Y，就称该实例为规范格式的。矩形的方法，当输入是规范格式时，总是返回规范格式的输出。

```
var ZR Rectangle
```

ZR是矩形的零值。

func Rect

```
func Rect(x0, y0, x1, y1 int) Rectangle
```

返回一个矩形Rectangle{Pt(x0, y0), Pt(x1, y1)}。

func (Rectangle) Canon

```
func (r Rectangle) Canon() Rectangle
```

返回矩形的规范版本（左上&右下），方法必要时会交换坐标的最大值和最小值。

func (Rectangle) Dx

```
func (r Rectangle) Dx() int
```

返回r的宽度。

func (Rectangle) Dy

```
func (r Rectangle) Dy() int
```

返回r的高度。

func (Rectangle) Size

```
func (r Rectangle) Size() Point
```

返回r的宽度w和高度h构成的点Point{w, h}。

func (Rectangle) Empty

```
func (r Rectangle) Empty() bool
```

报告矩形是否为空矩形。（即内部不包含点的矩形）

func (Rectangle) Eq

```
func (r Rectangle) Eq(s Rectangle) bool
```

报告两个矩形是否相同。

func (Rectangle) In

```
func (r Rectangle) In(s Rectangle) bool
```

如果r包含的所有点都在s内，则返回真；否则返回假。

func (Rectangle) Overlaps

```
func (r Rectangle) Overlaps(s Rectangle) bool
```

如果r和s有非空的交集，则返回真；否则返回假。

func (Rectangle) Add

```
func (r Rectangle) Add(p Point) Rectangle
```

返回矩形按p（作为向量）平移后的新矩形。

func (Rectangle) Sub

```
func (r Rectangle) Sub(p Point) Rectangle
```

返回矩形按p（作为向量）反向平移后的新矩形。

func (Rectangle) Intersect

```
func (r Rectangle) Intersect(s Rectangle) Rectangle
```

返回两个矩形的交集矩形（同时被r和s包含的最大矩形）；如果r和s没有重叠会返回Rectangle零值。

func (Rectangle) Union

```
func (r Rectangle) Union(s Rectangle) Rectangle
```

返回同时包含r和s的最小矩形。

func (Rectangle) Inset

```
func (r Rectangle) Inset(n int) Rectangle
```

返回去掉矩形四周宽度n的框的矩形，n可为负数。如果n过大将返回靠近r中心位置的空矩形。

func (Rectangle) String

```
func (r Rectangle) String() string
```

返回矩形的字符串表示，格式为"(3,4)-(6,5)"。

type Uniform

```
type Uniform struct {
    C color.Color
}
```

Uniform类型代表一块面积无限大的具有同一色彩的图像。它实现了color.Color、color.Model和Image等接口。

func NewUniform

```
func NewUniform(c color.Color) *Uniform
```

func (*Uniform) At

```
func (c *Uniform) At(x, y int) color.Color
```

func (*Uniform) Bounds

```
func (c *Uniform) Bounds() Rectangle
```

func (*Uniform) ColorModel

```
func (c *Uniform) ColorModel() color.Model
```

func (*Uniform) Convert

```
func (c *Uniform) Convert(color.Color) color.Color
```

func (*Uniform) Opaque

```
func (c *Uniform) Opaque() bool
```

Opaque方法扫描整个图像并报告该图像是否是完全不透明的。

func (*Uniform) RGBA

```
func (c *Uniform) RGBA() (r, g, b, a uint32)
```

type Alpha

```
type Alpha struct {
    // Pix保管图像的像素，内容为alpha通道值（即透明度）。
    // 像素(x, y)起始位置是Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)
    Pix []uint8
    // Stride是Pix中每行像素占用的字节数
    Stride int
    // Rect是图像的范围
    Rect Rectangle
}
```

Alpha类型代表一幅内存中的图像，其At方法返回color.Alpha类型的值。

func NewAlpha

```
func NewAlpha(r Rectangle) *Alpha
```

NewAlpha函数创建并返回一个具有指定宽度和高度的Alpha。

func (*Alpha) At

```
func (p *Alpha) At(x, y int) color.Color
```

func (*Alpha) Bounds

```
func (p *Alpha) Bounds() Rectangle
```

func (*Alpha) ColorModel

```
func (p *Alpha) ColorModel() color.Model
```

func (*Alpha) Opaque

```
func (p *Alpha) Opaque() bool
```

Opaque方法扫描整个图像并报告图像是否是完全不透明的。

func (*Alpha) PixOffset

```
func (p *Alpha) PixOffset(x, y int) int
```

PixOffset方法返回像素(x, y)的数据起始位置在Pix字段的偏移量/索引。

func (*Alpha) Set

```
func (p *Alpha) Set(x, y int, c color.Color)
```

func (*Alpha) SetAlpha

```
func (p *Alpha) SetAlpha(x, y int, c color.Alpha)
```

func (*Alpha) SubImage

```
func (p *Alpha) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

type Alpha16

```
type Alpha16 struct {
    // Pix保管图像的像素，内容为alpha通道值（即透明度，大端在前的格式）。
    // 像素(x, y)起始位置是Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)
    Pix []uint8
    // Stride是Pix中每行像素占用的字节数
    Stride int
    // Rect是图像的范围
    Rect Rectangle
}
```

Alpha16类型代表一幅内存中的图像，其At方法返回color.Alpha16类型的值。

func NewAlpha16

```
func NewAlpha16(r Rectangle) *Alpha16
```

NewAlpha16函数创建并返回一个具有指定范围的Alpha16。

func (*Alpha16) At

```
func (p *Alpha16) At(x, y int) color.Color
```

func (*Alpha16) Bounds

```
func (p *Alpha16) Bounds() Rectangle
```

func (*Alpha16) ColorModel

```
func (p *Alpha16) ColorModel() color.Model
```

func (*Alpha16) Opaque

```
func (p *Alpha16) Opaque() bool
```

Opaque方法扫描整个图像并报告图像是否是完全不透明的。

func (*Alpha16) PixOffset

```
func (p *Alpha16) PixOffset(x, y int) int
```

PixOffset方法返回像素(x, y)的数据起始位置在Pix字段的偏移量/索引。

func (*Alpha16) Set

```
func (p *Alpha16) Set(x, y int, c color.Color)
```

func (*Alpha16) SetAlpha16

```
func (p *Alpha16) SetAlpha16(x, y int, c color.Alpha16)
```

func (*Alpha16) SubImage

```
func (p *Alpha16) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

type Gray

```
type Gray struct {
    // Pix保管图像的像素，内容为灰度。
    // 像素(x, y)起始位置是Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)
    Pix []uint8
    // Stride是Pix中每行像素占用的字节数
    Stride int
    // Rect是图像的范围
    Rect Rectangle
}
```

Gray类型代表一幅内存中的图像，其At方法返回color.Gray类型的值。

func NewGray

```
func NewGray(r Rectangle) *Gray
```

NewGray函数创建并返回一个具有指定范围的Gray。

func (*Gray) At

```
func (p *Gray) At(x, y int) color.Color
```

func (*Gray) Bounds

```
func (p *Gray) Bounds() Rectangle
```

func (*Gray) ColorModel

```
func (p *Gray) ColorModel() color.Model
```

func (*Gray) Opaque

```
func (p *Gray) Opaque() bool
```

Opaque方法扫描整个图像并报告图像是否是完全不透明的。

func (*Gray) PixOffset

```
func (p *Gray) PixOffset(x, y int) int
```

PixOffset方法返回像素(x, y)的数据起始位置在Pix字段的偏移量/索引。

func (*Gray) Set

```
func (p *Gray) Set(x, y int, c color.Color)
```

func (*Gray) SetGray

```
func (p *Gray) SetGray(x, y int, c color.Gray)
```

func (*Gray) SubImage

```
func (p *Gray) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

type Gray16

```
type Gray16 struct {
    // Pix保管图像的像素，内容为灰度（大端在前的格式）。
    // 像素(x, y)起始位置是Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)
    Pix []uint8
    // Stride是Pix中每行像素占用的字节数
    Stride int
    // Rect是图像的范围
    Rect Rectangle
}
```

Gray16类型代表一幅内存中的图像，其At方法返回color.Gray16类型的值。

func NewGray16

```
func NewGray16(r Rectangle) *Gray16
```

NewGray16函数创建并返回一个具有指定范围的Gray16。

func (*Gray16) At

```
func (p *Gray16) At(x, y int) color.Color
```

func (*Gray16) Bounds

```
func (p *Gray16) Bounds() Rectangle
```

func (*Gray16) ColorModel

```
func (p *Gray16) ColorModel() color.Model
```

func (*Gray16) Opaque

```
func (p *Gray16) Opaque() bool
```

Opaque方法扫描整个图像并报告图像是否是完全不透明的。

func (*Gray16) PixOffset

```
func (p *Gray16) PixOffset(x, y int) int
```

PixOffset方法返回像素(x, y)的数据起始位置在Pix字段的偏移量/索引。

func (*Gray16) Set

```
func (p *Gray16) Set(x, y int, c color.Color)
```

func (*Gray16) SetGray16

```
func (p *Gray16) SetGray16(x, y int, c color.Gray16)
```

func (*Gray16) SubImage

```
func (p *Gray16) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

type RGBA

```
type RGBA struct {
    // Pix保管图像的像素色彩信息，顺序为R, G, B, A
    // 像素(x, y)起始位置是Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)
    Pix []uint8
    // Stride是Pix中每行像素占用的字节数
    Stride int
    // Rect是图像的范围
    Rect Rectangle
}
```

RGBA类型代表一幅内存中的图像，其At方法返回color.RGBA类型的值。

func NewRGBA

```
func NewRGBA(r Rectangle) *RGBA
```

NewRGBA函数创建并返回一个具有指定范围的RGBA。

func (*RGBA) At

```
func (p *RGBA) At(x, y int) color.Color
```

func (*RGBA) Bounds

```
func (p *RGBA) Bounds() Rectangle
```

func (*RGBA) ColorModel

```
func (p *RGBA) ColorModel() color.Model
```

func (*RGBA) Opaque

```
func (p *RGBA) Opaque() bool
```

Opaque方法扫描整个图像并报告图像是否是完全不透明的。

func (*RGBA) PixOffset

```
func (p *RGBA) PixOffset(x, y int) int
```

PixOffset方法返回像素(x, y)的数据起始位置在Pix字段的偏移量/索引。

func (*RGBA) Set

```
func (p *RGBA) Set(x, y int, c color.Color)
```

func (*RGBA) SetRGBA

```
func (p *RGBA) SetRGBA(x, y int, c color.RGBA)
```

func (*RGBA) SubImage

```
func (p *RGBA) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

type RGBA64

```
type RGBA64 struct {
    // Pix保管图像的像素色彩信息，顺序为R, G, B, A（每个值都是大端在前的格式
    // 像素(x, y)起始位置是Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)
    Pix []uint8
    // Stride是Pix中每行像素占用的字节数
    Stride int
    // Rect是图像的范围
    Rect Rectangle
}
```

RGBA64类型代表一幅内存中的图像，其At方法返回color.RGBA64类型的值

func NewRGBA64

```
func NewRGBA64(r Rectangle) *RGBA64
```

NewRGBA64函数创建并返回一个具有指定范围的RGBA64

func (*RGBA64) At

```
func (p *RGBA64) At(x, y int) color.Color
```

func (*RGBA64) Bounds

```
func (p *RGBA64) Bounds() Rectangle
```

func (*RGBA64) ColorModel

```
func (p *RGBA64) ColorModel() color.Model
```

func (*RGBA64) Opaque

```
func (p *RGBA64) Opaque() bool
```

Opaque方法扫描整个图像并报告图像是否是完全不透明的。

func (*RGBA64) PixOffset

```
func (p *RGBA64) PixOffset(x, y int) int
```

PixOffset方法返回像素(x, y)的数据起始位置在Pix字段的偏移量/索引。

func (*RGBA64) Set

```
func (p *RGBA64) Set(x, y int, c color.Color)
```

func (*RGBA64) SetRGBA64

```
func (p *RGBA64) SetRGBA64(x, y int, c color.RGBA64)
```

func (*RGBA64) SubImage

```
func (p *RGBA64) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

type NRGBA

```
type NRGBA struct {
    // Pix保管图像的像素色彩信息，顺序为R, G, B, A
    // 像素(x, y)起始位置是Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)
    Pix []uint8
    // Stride是Pix中每行像素占用的字节数
    Stride int
    // Rect是图像的范围
    Rect Rectangle
}
```

NRGBA类型代表一幅内存中的图像，其At方法返回color.NRGBA类型的值。

func NewNRGBA

```
func NewNRGBA(r Rectangle) *NRGBA
```

NewNRGBA函数创建并返回一个具有指定范围的NRGBA。

func (*NRGBA) At

```
func (p *NRGBA) At(x, y int) color.Color
```

func (*NRGBA) Bounds

```
func (p *NRGBA) Bounds() Rectangle
```

func (*NRGBA) ColorModel

```
func (p *NRGBA) ColorModel() color.Model
```

func (*NRGBA) Opaque

```
func (p *NRGBA) Opaque() bool
```

Opaque方法扫描整个图像并报告图像是否是完全不透明的。

func (*NRGBA) PixOffset

```
func (p *NRGBA) PixOffset(x, y int) int
```

PixOffset方法返回像素(x, y)的数据起始位置在Pix字段的偏移量/索引。

func (*NRGBA) Set

```
func (p *NRGBA) Set(x, y int, c color.Color)
```

func (*NRGBA) SetNRGBA

```
func (p *NRGBA) SetNRGBA(x, y int, c color.NRGBA)
```

func (*NRGBA) SubImage

```
func (p *NRGBA) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

type NRGBA64

```
type NRGBA64 struct {
    // Pix保管图像的像素色彩信息，顺序为R, G, B, A（每个值都是大端在前的格式
    // 像素(x, y)起始位置是Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)
    Pix []uint8
    // Stride是Pix中每行像素占用的字节数
    Stride int
    // Rect是图像的范围
    Rect Rectangle
}
```

NRGBA64类型代表一幅内存中的图像，其At方法返回color.NRGBA64类型的值。

func NewNRGBA64

```
func NewNRGBA64(r Rectangle) *NRGBA64
```

NewNRGBA64函数创建并返回一个具有指定范围的NRGBA64。

func (*NRGBA64) At

```
func (p *NRGBA64) At(x, y int) color.Color
```

func (*NRGBA64) Bounds

```
func (p *NRGBA64) Bounds() Rectangle
```

func (*NRGBA64) ColorModel

```
func (p *NRGBA64) ColorModel() color.Model
```

func (*NRGBA64) Opaque

```
func (p *NRGBA64) Opaque() bool
```

Opaque方法扫描整个图像并报告图像是否是完全不透明的。

func (*NRGBA64) PixOffset

```
func (p *NRGBA64) PixOffset(x, y int) int
```

PixOffset方法返回像素(x, y)的数据起始位置在Pix字段的偏移量/索引。

func (*NRGBA64) Set

```
func (p *NRGBA64) Set(x, y int, c color.Color)
```

func (*NRGBA64) SetNRGBA64

```
func (p *NRGBA64) SetNRGBA64(x, y int, c color.NRGBA64)
```

func (*NRGBA64) SubImage

```
func (p *NRGBA64) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

type Paletted

```
type Paletted struct {
    // Pix保存图像的象素，内容为调色板的索引。
    // 像素(x, y)的位置是Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*1
    Pix []uint8
    // Stride是Pix中每行像素占用的字节数
    Stride int
    // Rect是图像的范围
    Rect Rectangle
    // Palette是图像的调色板
    Palette color.Palette
}
```

Paletted类型是一幅采用uint8类型索引调色板的内存中的图像。

func NewPaletted

```
func NewPaletted(r Rectangle, p color.Palette) *Paletted
```

NewPaletted函数创建并返回一个具有指定范围、调色板的Paletted。

func (*Paletted) At

```
func (p *Paletted) At(x, y int) color.Color
```

func (*Paletted) Bounds

```
func (p *Paletted) Bounds() Rectangle
```

func (*Paletted) ColorIndexAt

```
func (p *Paletted) ColorIndexAt(x, y int) uint8
```

func (*Paletted) ColorModel

```
func (p *Paletted) ColorModel() color.Model
```

func (*Paletted) Opaque

```
func (p *Paletted) Opaque() bool
```

Opaque方法扫描整个图像并报告图像是否是完全不透明的。

func (*Paletted) PixOffset

```
func (p *Paletted) PixOffset(x, y int) int
```

PixOffset方法返回像素(x, y)的数据起始位置在Pix字段的偏移量/索引。

func (*Paletted) Set

```
func (p *Paletted) Set(x, y int, c color.Color)
```

func (*Paletted) SetColorIndex

```
func (p *Paletted) SetColorIndex(x, y int, index uint8)
```

func (*Paletted) SubImage

```
func (p *Paletted) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

type YCbCrSubsampleRatio

```
type YCbCrSubsampleRatio int
```

YcbCrSubsampleRatio是YCbCr图像的色度二次采样比率。

```
const (
    YCbCrSubsampleRatio444 YCbCrSubsampleRatio = iota
    YCbCrSubsampleRatio422
    YCbCrSubsampleRatio420
    YCbCrSubsampleRatio440
)
```

func (YCbCrSubsampleRatio) String

```
func (s YCbCrSubsampleRatio) String() string
```

type YCbCr

```
type YCbCr struct {
    Y, Cb, Cr      []uint8
    YStride        int
    CStride        int
    SubsampleRatio YCbCrSubsampleRatio
    Rect           Rectangle
}
```

YcbCr代表采用Y'CbCr色彩模型的一幅内存中的图像。每个像素都对应一个Y采样，但每个Cb/Cr采样对应多个像素。Ystride是两个垂直相邻的像素之间的Y组分的索引增量。CStride是两个映射到单独的色度采样的垂直相邻的像素之间的Cb/Cr组分的索引增量。虽然不作绝对要求，但Ystride字段和len(Y)一般应为8的倍数，并且：

```
For 4:4:4, CStride == YStride/1 && len(Cb) == len(Cr) == len(Y)/1.
For 4:2:2, CStride == YStride/2 && len(Cb) == len(Cr) == len(Y)/2.
For 4:2:0, CStride == YStride/2 && len(Cb) == len(Cr) == len(Y)/4.
For 4:4:0, CStride == YStride/1 && len(Cb) == len(Cr) == len(Y)/2.
```

func NewYCbCr

```
func NewYCbCr(r Rectangle, subsampleRatio YCbCrSubsampleRatio) *YCb
```

NewYCbCr函数创建并返回一个具有指定宽度、高度和二次采样率的YcbCr。

func (*YCbCr) At

```
func (p *YCbCr) At(x, y int) color.Color
```

func (*YCbCr) Bounds

```
func (p *YCbCr) Bounds() Rectangle
```

func (*YCbCr) ColorModel

```
func (p *YCbCr) ColorModel() color.Model
```

func (*YCbCr) Opaque

```
func (p *YCbCr) Opaque() bool
```

func (*YCbCr) COffset

```
func (p *YCbCr) COffset(x, y int) int
```

像素(X, Y)的Cb或Cr（色度）组分的数据起始位置在Cb/Cr字段的偏移量/索引。

func (*YCbCr) YOffset

```
func (p *YCbCr) YOffset(x, y int) int
```

像素(X, Y)的Y（亮度）组分的数据起始位置在Y字段的偏移量/索引。

func (*YCbCr) SubImage

```
func (p *YCbCr) SubImage(r Rectangle) Image
```

SubImage方法返回代表原图像一部分（r的范围）的新图像。返回值和原图像的像素数据是共用的。

package color

```
import "image/color"
```

color包实现了基本色彩库。

Index

Variables

```
var (
    Black       = Gray16{0}       // 黑色
    White       = Gray16{0xffff}  // 白色
    Transparent = Alpha16{0}      // 完全透明
    Opaque      = Alpha16{0xffff} // 完全不透明
)
```

标准色彩。

type Color

```
type Color interface {
    // 方法返回预乘了alpha的红、绿、蓝色彩值和alpha通道值，范围都在[0, 0xFF
    // 返回值类型为uint32，这样当乘以接近0xFFFF的混合参数时，不会溢出。
    RGBA() (r, g, b, a uint32)
}
```

实现了Color接口的类型可以将自身转化为预乘了alpha的16位通道的RGBA，转换可能会丢失色彩信息。

type Model

```
type Model interface {
    Convert(c Color) Color
}
```

Model接口可以将任意Color接口转换为采用自身色彩模型的Color接口。转换可能会丢失色彩信息。

```
var (
    RGBAModel    Model = ModelFunc(rgbaModel)
    RGBA64Model  Model = ModelFunc(rgba64Model)
    NRGBAModel   Model = ModelFunc(nrgbaModel)
    NRGBA64Model Model = ModelFunc(nrgba64Model)
    AlphaModel   Model = ModelFunc(alphaModel)
    Alpha16Model Model = ModelFunc(alpha16Model)
    GrayModel    Model = ModelFunc(grayModel)
    Gray16Model  Model = ModelFunc(gray16Model)
)
```

Models接口返回标准的Color接口类型。

```
var YCbCrModel Model = ModelFunc(yCbCrModel)
```

包变量YcbCrModel是Y'cbCr色彩模型的Model接口。

func ModelFunc

```
func ModelFunc(f func(Color) Color) Model
```

函数ModelFunc返回一个调用函数f实现色彩转换的Model接口。

type Alpha

```
type Alpha struct {
    A uint8
}
```

Alpha类型代表一个8位的alpha通道（alpha通道表示透明度）。

func (Alpha) RGBA

```
func (c Alpha) RGBA() (r, g, b, a uint32)
```

type Alpha16

```
type Alpha16 struct {
    A uint16
}
```

Alpha16类型代表一个16位的alpha通道。

func (Alpha16) RGBA

```
func (c Alpha16) RGBA() (r, g, b, a uint32)
```

type Gray

```
type Gray struct {
    Y uint8
}
```

Gray类型代表一个8位的灰度色彩。

func (Gray) RGBA

```
func (c Gray) RGBA() (r, g, b, a uint32)
```

type Gray16

```
type Gray16 struct {
    Y uint16
}
```

Gray16类型代表一个16位的灰度色彩。

func (Gray16) RGBA

```
func (c Gray16) RGBA() (r, g, b, a uint32)
```

type RGBA

```
type RGBA struct {
    R, G, B, A uint8
}
```

RGBA类型代表传统的预乘了alpha通道的32位RGB色彩，Red、Green、Blue、Alpha各8位。

func (RGBA) RGBA

```
func (c RGBA) RGBA() (r, g, b, a uint32)
```

type RGBA64

```
type RGBA64 struct {
    R, G, B, A uint16
}
```

RGBA64类型代表预乘了alpha通道的64位RGB色彩，Red、Green、Blue、Alpha各16位。

func (RGBA64) RGBA

```
func (c RGBA64) RGBA() (r, g, b, a uint32)
```

type NRGBA

```
type NRGBA struct {
    R, G, B, A uint8
}
```

NRGBA类型代表没有预乘alpha通道的32位RGB色彩，Red、Green、Blue、Alpha各8位。

func (NRGBA) RGBA

```
func (c NRGBA) RGBA() (r, g, b, a uint32)
```

type NRGBA64

```
type NRGBA64 struct {
    R, G, B, A uint16
}
```

NRGBA64类型代表没有预乘alpha通道的64位RGB色彩，Red、Green、Blue、Alpha各16位。

func (NRGBA64) RGBA

```
func (c NRGBA64) RGBA() (r, g, b, a uint32)
```

type YCbCr

```
type YCbCr struct {
    Y, Cb, Cr uint8
}
```

YcbCr代表完全不透明的24位Y'CbCr色彩；每个色彩都有1个亮度成分和2个色度成分，分别用8位字节表示。

JPEG、VP8、MPEG家族和其他编码方式使用本色彩模型。这些编码通常将Y'CbCr 和YUV两个色彩模型等同使用（Y=Y'=黄、U=Cb=青、V=Cr=品红）。但严格来说，YUV模只用于模拟视频信号，Y'是经过伽玛校正的Y。RGB和Y'CbCr色彩模型之间的转换会丢失色彩信息。两个色彩模型之间的转换有多个存在细微区别的算法。本包采用JFIF算法，参见http://www.w3.org/Graphics/JPEG/jfif3.pdf

func (YCbCr) RGBA

```
func (c YCbCr) RGBA() (uint32, uint32, uint32, uint32)
```

type Palette

```
type Palette []Color
```

Palette类型代表一个色彩的调色板。

func (Palette) Convert

```
func (p Palette) Convert(c Color) Color
```

返回调色板中与色彩c在欧几里德RGB色彩空间最接近的色彩。（实现了Model接口）

func (Palette) Index

```
func (p Palette) Index(c Color) int
```

返回调色板中与色彩c在欧几里德RGB色彩空间最接近的色彩的索引。

func RGBToYCbCr

```
func RGBToYCbCr(r, g, b uint8) (uint8, uint8, uint8)
```

函数将RGB三原色转换为Y'CbCr三原色。

func YCbCrToRGB

```
func YCbCrToRGB(y, cb, cr uint8) (uint8, uint8, uint8)
```

函数将Y'CbCr三原色转换为RGB三原色。

package palette

```
import "image/color/palette"
```

palette包提供了标准的调色板。

Index

Variables

```
var Plan9 = []color.Color{
    color.RGBA{0x00, 0x00, 0x00, 0xff},
    color.RGBA{0x00, 0x00, 0x44, 0xff},
    color.RGBA{0x00, 0x00, 0x88, 0xff},
    color.RGBA{0x00, 0x00, 0xcc, 0xff},
    color.RGBA{0x00, 0x44, 0x00, 0xff},
    color.RGBA{0x00, 0x44, 0x44, 0xff},
    color.RGBA{0x00, 0x44, 0x88, 0xff},
    color.RGBA{0x00, 0x44, 0xcc, 0xff},
    color.RGBA{0x00, 0x88, 0x00, 0xff},
    color.RGBA{0x00, 0x88, 0x44, 0xff},
    color.RGBA{0x00, 0x88, 0x88, 0xff},
    color.RGBA{0x00, 0x88, 0xcc, 0xff},
    color.RGBA{0x00, 0xcc, 0x00, 0xff},
    color.RGBA{0x00, 0xcc, 0x44, 0xff},
    color.RGBA{0x00, 0xcc, 0x88, 0xff},
    color.RGBA{0x00, 0xcc, 0xcc, 0xff},
    color.RGBA{0x00, 0xdd, 0xdd, 0xff},
    color.RGBA{0x11, 0x11, 0x11, 0xff},
    color.RGBA{0x00, 0x00, 0x55, 0xff},
    color.RGBA{0x00, 0x00, 0x99, 0xff},
    color.RGBA{0x00, 0x00, 0xdd, 0xff},
    color.RGBA{0x00, 0x55, 0x00, 0xff},
    color.RGBA{0x00, 0x55, 0x55, 0xff},
    color.RGBA{0x00, 0x4c, 0x99, 0xff},
    color.RGBA{0x00, 0x49, 0xdd, 0xff},
    color.RGBA{0x00, 0x99, 0x00, 0xff},
    color.RGBA{0x00, 0x99, 0x4c, 0xff},
    color.RGBA{0x00, 0x99, 0x99, 0xff},
    color.RGBA{0x00, 0x93, 0xdd, 0xff},
    color.RGBA{0x00, 0xdd, 0x00, 0xff},
    color.RGBA{0x00, 0xdd, 0x49, 0xff},
    color.RGBA{0x00, 0xdd, 0x93, 0xff},
    color.RGBA{0x00, 0xee, 0x9e, 0xff},
```

```
    color.RGBA{0x00, 0xee, 0xee, 0xff},
    color.RGBA{0x22, 0x22, 0x22, 0xff},
    color.RGBA{0x00, 0x00, 0x66, 0xff},
    color.RGBA{0x00, 0x00, 0xaa, 0xff},
    color.RGBA{0x00, 0x00, 0xee, 0xff},
    color.RGBA{0x00, 0x66, 0x00, 0xff},
    color.RGBA{0x00, 0x66, 0x66, 0xff},
    color.RGBA{0x00, 0x55, 0xaa, 0xff},
    color.RGBA{0x00, 0x4f, 0xee, 0xff},
    color.RGBA{0x00, 0xaa, 0x00, 0xff},
    color.RGBA{0x00, 0xaa, 0x55, 0xff},
    color.RGBA{0x00, 0xaa, 0xaa, 0xff},
    color.RGBA{0x00, 0x9e, 0xee, 0xff},
    color.RGBA{0x00, 0xee, 0x00, 0xff},
    color.RGBA{0x00, 0xee, 0x4f, 0xff},
    color.RGBA{0x00, 0xff, 0x55, 0xff},
    color.RGBA{0x00, 0xff, 0xaa, 0xff},
    color.RGBA{0x00, 0xff, 0xff, 0xff},
    color.RGBA{0x33, 0x33, 0x33, 0xff},
    color.RGBA{0x00, 0x00, 0x77, 0xff},
    color.RGBA{0x00, 0x00, 0xbb, 0xff},
    color.RGBA{0x00, 0x00, 0xff, 0xff},
    color.RGBA{0x00, 0x77, 0x00, 0xff},
    color.RGBA{0x00, 0x77, 0x77, 0xff},
    color.RGBA{0x00, 0x5d, 0xbb, 0xff},
    color.RGBA{0x00, 0x55, 0xff, 0xff},
    color.RGBA{0x00, 0xbb, 0x00, 0xff},
    color.RGBA{0x00, 0xbb, 0x5d, 0xff},
    color.RGBA{0x00, 0xbb, 0xbb, 0xff},
    color.RGBA{0x00, 0xaa, 0xff, 0xff},
    color.RGBA{0x00, 0xff, 0x00, 0xff},
    color.RGBA{0x44, 0x00, 0x44, 0xff},
    color.RGBA{0x44, 0x00, 0x88, 0xff},
    color.RGBA{0x44, 0x00, 0xcc, 0xff},
    color.RGBA{0x44, 0x44, 0x00, 0xff},
    color.RGBA{0x44, 0x44, 0x44, 0xff},
    color.RGBA{0x44, 0x44, 0x88, 0xff},
    color.RGBA{0x44, 0x44, 0xcc, 0xff},
    color.RGBA{0x44, 0x88, 0x00, 0xff},
    color.RGBA{0x44, 0x88, 0x44, 0xff},
    color.RGBA{0x44, 0x88, 0x88, 0xff},
    color.RGBA{0x44, 0x88, 0xcc, 0xff},
    color.RGBA{0x44, 0xcc, 0x00, 0xff},
    color.RGBA{0x44, 0xcc, 0x44, 0xff},
    color.RGBA{0x44, 0xcc, 0x88, 0xff},
    color.RGBA{0x44, 0xcc, 0xcc, 0xff},
    color.RGBA{0x44, 0x00, 0x00, 0xff},
    color.RGBA{0x55, 0x00, 0x00, 0xff},
    color.RGBA{0x55, 0x00, 0x55, 0xff},
    color.RGBA{0x4c, 0x00, 0x99, 0xff},
    color.RGBA{0x49, 0x00, 0xdd, 0xff},
    color.RGBA{0x55, 0x55, 0x00, 0xff},
    color.RGBA{0x55, 0x55, 0x55, 0xff},
```

```
        color.RGBA{0x4c, 0x4c, 0x99, 0xff},
        color.RGBA{0x49, 0x49, 0xdd, 0xff},
        color.RGBA{0x4c, 0x99, 0x00, 0xff},
        color.RGBA{0x4c, 0x99, 0x4c, 0xff},
        color.RGBA{0x4c, 0x99, 0x99, 0xff},
        color.RGBA{0x49, 0x93, 0xdd, 0xff},
        color.RGBA{0x49, 0xdd, 0x00, 0xff},
        color.RGBA{0x49, 0xdd, 0x49, 0xff},
        color.RGBA{0x49, 0xdd, 0x93, 0xff},
        color.RGBA{0x49, 0xdd, 0xdd, 0xff},
        color.RGBA{0x4f, 0xee, 0xee, 0xff},
        color.RGBA{0x66, 0x00, 0x00, 0xff},
        color.RGBA{0x66, 0x00, 0x66, 0xff},
        color.RGBA{0x55, 0x00, 0xaa, 0xff},
        color.RGBA{0x4f, 0x00, 0xee, 0xff},
        color.RGBA{0x66, 0x66, 0x00, 0xff},
        color.RGBA{0x66, 0x66, 0x66, 0xff},
        color.RGBA{0x55, 0x55, 0xaa, 0xff},
        color.RGBA{0x4f, 0x4f, 0xee, 0xff},
        color.RGBA{0x55, 0xaa, 0x00, 0xff},
        color.RGBA{0x55, 0xaa, 0x55, 0xff},
        color.RGBA{0x55, 0xaa, 0xaa, 0xff},
        color.RGBA{0x4f, 0x9e, 0xee, 0xff},
        color.RGBA{0x4f, 0xee, 0x00, 0xff},
        color.RGBA{0x4f, 0xee, 0x4f, 0xff},
        color.RGBA{0x4f, 0xee, 0x9e, 0xff},
        color.RGBA{0x55, 0xff, 0xaa, 0xff},
        color.RGBA{0x55, 0xff, 0xff, 0xff},
        color.RGBA{0x77, 0x00, 0x00, 0xff},
        color.RGBA{0x77, 0x00, 0x77, 0xff},
        color.RGBA{0x5d, 0x00, 0xbb, 0xff},
        color.RGBA{0x55, 0x00, 0xff, 0xff},
        color.RGBA{0x77, 0x77, 0x00, 0xff},
        color.RGBA{0x77, 0x77, 0x77, 0xff},
        color.RGBA{0x5d, 0x5d, 0xbb, 0xff},
        color.RGBA{0x55, 0x55, 0xff, 0xff},
        color.RGBA{0x5d, 0xbb, 0x00, 0xff},
        color.RGBA{0x5d, 0xbb, 0x5d, 0xff},
        color.RGBA{0x5d, 0xbb, 0xbb, 0xff},
        color.RGBA{0x55, 0xaa, 0xff, 0xff},
        color.RGBA{0x55, 0xff, 0x00, 0xff},
        color.RGBA{0x55, 0xff, 0x55, 0xff},
        color.RGBA{0x88, 0x00, 0x88, 0xff},
        color.RGBA{0x88, 0x00, 0xcc, 0xff},
        color.RGBA{0x88, 0x44, 0x00, 0xff},
        color.RGBA{0x88, 0x44, 0x44, 0xff},
        color.RGBA{0x88, 0x44, 0x88, 0xff},
        color.RGBA{0x88, 0x44, 0xcc, 0xff},
        color.RGBA{0x88, 0x88, 0x00, 0xff},
        color.RGBA{0x88, 0x88, 0x44, 0xff},
        color.RGBA{0x88, 0x88, 0x88, 0xff},
        color.RGBA{0x88, 0x88, 0xcc, 0xff},
        color.RGBA{0x88, 0xcc, 0x00, 0xff},
```

```
        color.RGBA{0x88, 0xcc, 0x44, 0xff},
        color.RGBA{0x88, 0xcc, 0x88, 0xff},
        color.RGBA{0x88, 0xcc, 0xcc, 0xff},
        color.RGBA{0x88, 0x00, 0x00, 0xff},
        color.RGBA{0x88, 0x00, 0x44, 0xff},
        color.RGBA{0x99, 0x00, 0x4c, 0xff},
        color.RGBA{0x99, 0x00, 0x99, 0xff},
        color.RGBA{0x93, 0x00, 0xdd, 0xff},
        color.RGBA{0x99, 0x4c, 0x00, 0xff},
        color.RGBA{0x99, 0x4c, 0x4c, 0xff},
        color.RGBA{0x99, 0x4c, 0x99, 0xff},
        color.RGBA{0x93, 0x49, 0xdd, 0xff},
        color.RGBA{0x99, 0x99, 0x00, 0xff},
        color.RGBA{0x99, 0x99, 0x4c, 0xff},
        color.RGBA{0x99, 0x99, 0x99, 0xff},
        color.RGBA{0x93, 0x93, 0xdd, 0xff},
        color.RGBA{0x93, 0xdd, 0x00, 0xff},
        color.RGBA{0x93, 0xdd, 0x49, 0xff},
        color.RGBA{0x93, 0xdd, 0x93, 0xff},
        color.RGBA{0x93, 0xdd, 0xdd, 0xff},
        color.RGBA{0x99, 0x00, 0x00, 0xff},
        color.RGBA{0xaa, 0x00, 0x00, 0xff},
        color.RGBA{0xaa, 0x00, 0x55, 0xff},
        color.RGBA{0xaa, 0x00, 0xaa, 0xff},
        color.RGBA{0x9e, 0x00, 0xee, 0xff},
        color.RGBA{0xaa, 0x55, 0x00, 0xff},
        color.RGBA{0xaa, 0x55, 0x55, 0xff},
        color.RGBA{0xaa, 0x55, 0xaa, 0xff},
        color.RGBA{0x9e, 0x4f, 0xee, 0xff},
        color.RGBA{0xaa, 0xaa, 0x00, 0xff},
        color.RGBA{0xaa, 0xaa, 0x55, 0xff},
        color.RGBA{0xaa, 0xaa, 0xaa, 0xff},
        color.RGBA{0x9e, 0x9e, 0xee, 0xff},
        color.RGBA{0x9e, 0xee, 0x00, 0xff},
        color.RGBA{0x9e, 0xee, 0x4f, 0xff},
        color.RGBA{0x9e, 0xee, 0x9e, 0xff},
        color.RGBA{0x9e, 0xee, 0xee, 0xff},
        color.RGBA{0xaa, 0xff, 0xff, 0xff},
        color.RGBA{0xbb, 0x00, 0x00, 0xff},
        color.RGBA{0xbb, 0x00, 0x5d, 0xff},
        color.RGBA{0xbb, 0x00, 0xbb, 0xff},
        color.RGBA{0xaa, 0x00, 0xff, 0xff},
        color.RGBA{0xbb, 0x5d, 0x00, 0xff},
        color.RGBA{0xbb, 0x5d, 0x5d, 0xff},
        color.RGBA{0xbb, 0x5d, 0xbb, 0xff},
        color.RGBA{0xaa, 0x55, 0xff, 0xff},
        color.RGBA{0xbb, 0xbb, 0x00, 0xff},
        color.RGBA{0xbb, 0xbb, 0x5d, 0xff},
        color.RGBA{0xbb, 0xbb, 0xbb, 0xff},
        color.RGBA{0xaa, 0xaa, 0xff, 0xff},
        color.RGBA{0xaa, 0xff, 0x00, 0xff},
        color.RGBA{0xaa, 0xff, 0x55, 0xff},
        color.RGBA{0xaa, 0xff, 0xaa, 0xff},
```

```
    color.RGBA{0xcc, 0x00, 0xcc, 0xff},
    color.RGBA{0xcc, 0x44, 0x00, 0xff},
    color.RGBA{0xcc, 0x44, 0x44, 0xff},
    color.RGBA{0xcc, 0x44, 0x88, 0xff},
    color.RGBA{0xcc, 0x44, 0xcc, 0xff},
    color.RGBA{0xcc, 0x88, 0x00, 0xff},
    color.RGBA{0xcc, 0x88, 0x44, 0xff},
    color.RGBA{0xcc, 0x88, 0x88, 0xff},
    color.RGBA{0xcc, 0x88, 0xcc, 0xff},
    color.RGBA{0xcc, 0xcc, 0x00, 0xff},
    color.RGBA{0xcc, 0xcc, 0x44, 0xff},
    color.RGBA{0xcc, 0xcc, 0x88, 0xff},
    color.RGBA{0xcc, 0xcc, 0xcc, 0xff},
    color.RGBA{0xcc, 0x00, 0x00, 0xff},
    color.RGBA{0xcc, 0x00, 0x44, 0xff},
    color.RGBA{0xcc, 0x00, 0x88, 0xff},
    color.RGBA{0xdd, 0x00, 0x93, 0xff},
    color.RGBA{0xdd, 0x00, 0xdd, 0xff},
    color.RGBA{0xdd, 0x49, 0x00, 0xff},
    color.RGBA{0xdd, 0x49, 0x49, 0xff},
    color.RGBA{0xdd, 0x49, 0x93, 0xff},
    color.RGBA{0xdd, 0x49, 0xdd, 0xff},
    color.RGBA{0xdd, 0x93, 0x00, 0xff},
    color.RGBA{0xdd, 0x93, 0x49, 0xff},
    color.RGBA{0xdd, 0x93, 0x93, 0xff},
    color.RGBA{0xdd, 0x93, 0xdd, 0xff},
    color.RGBA{0xdd, 0xdd, 0x00, 0xff},
    color.RGBA{0xdd, 0xdd, 0x49, 0xff},
    color.RGBA{0xdd, 0xdd, 0x93, 0xff},
    color.RGBA{0xdd, 0xdd, 0xdd, 0xff},
    color.RGBA{0xdd, 0x00, 0x00, 0xff},
    color.RGBA{0xdd, 0x00, 0x49, 0xff},
    color.RGBA{0xee, 0x00, 0x4f, 0xff},
    color.RGBA{0xee, 0x00, 0x9e, 0xff},
    color.RGBA{0xee, 0x00, 0xee, 0xff},
    color.RGBA{0xee, 0x4f, 0x00, 0xff},
    color.RGBA{0xee, 0x4f, 0x4f, 0xff},
    color.RGBA{0xee, 0x4f, 0x9e, 0xff},
    color.RGBA{0xee, 0x4f, 0xee, 0xff},
    color.RGBA{0xee, 0x9e, 0x00, 0xff},
    color.RGBA{0xee, 0x9e, 0x4f, 0xff},
    color.RGBA{0xee, 0x9e, 0x9e, 0xff},
    color.RGBA{0xee, 0x9e, 0xee, 0xff},
    color.RGBA{0xee, 0xee, 0x00, 0xff},
    color.RGBA{0xee, 0xee, 0x4f, 0xff},
    color.RGBA{0xee, 0xee, 0x9e, 0xff},
    color.RGBA{0xee, 0xee, 0xee, 0xff},
    color.RGBA{0xee, 0x00, 0x00, 0xff},
    color.RGBA{0xff, 0x00, 0x00, 0xff},
    color.RGBA{0xff, 0x00, 0x55, 0xff},
    color.RGBA{0xff, 0x00, 0xaa, 0xff},
    color.RGBA{0xff, 0x00, 0xff, 0xff},
    color.RGBA{0xff, 0x55, 0x00, 0xff},
```

```
    color.RGBA{0xff, 0x55, 0x55, 0xff},
    color.RGBA{0xff, 0x55, 0xaa, 0xff},
    color.RGBA{0xff, 0x55, 0xff, 0xff},
    color.RGBA{0xff, 0xaa, 0x00, 0xff},
    color.RGBA{0xff, 0xaa, 0x55, 0xff},
    color.RGBA{0xff, 0xaa, 0xaa, 0xff},
    color.RGBA{0xff, 0xaa, 0xff, 0xff},
    color.RGBA{0xff, 0xff, 0x00, 0xff},
    color.RGBA{0xff, 0xff, 0x55, 0xff},
    color.RGBA{0xff, 0xff, 0xaa, 0xff},
    color.RGBA{0xff, 0xff, 0xff, 0xff},
}
```

Plan9是一个256色的调色板，将24位的RGB色彩立方划分为4x4x4的子立方，每个子立方体都有4个色调。比起WebSafe，本方案通过将颜色立方划分成更少的方格减少了颜色的分辨率，以便用额外的空间换取增加强度的分辨率。结果是灰色有16个色调（4个灰色子立方每个有4个采样），初级和次级颜色各有13个色调（3个子立方每个有4个采样，再加上黑色），加上对色彩立方其余部分的合理色彩选择，实现了更好的连续色调的表现。

本调色板用于Plan9操作系统，参见http://plan9.bell-labs.com/magic/man2html/6/color

```
var WebSafe = []color.Color{
    color.RGBA{0x00, 0x00, 0x00, 0xff},
    color.RGBA{0x00, 0x00, 0x33, 0xff},
    color.RGBA{0x00, 0x00, 0x66, 0xff},
    color.RGBA{0x00, 0x00, 0x99, 0xff},
    color.RGBA{0x00, 0x00, 0xcc, 0xff},
    color.RGBA{0x00, 0x00, 0xff, 0xff},
    color.RGBA{0x00, 0x33, 0x00, 0xff},
    color.RGBA{0x00, 0x33, 0x33, 0xff},
    color.RGBA{0x00, 0x33, 0x66, 0xff},
    color.RGBA{0x00, 0x33, 0x99, 0xff},
    color.RGBA{0x00, 0x33, 0xcc, 0xff},
    color.RGBA{0x00, 0x33, 0xff, 0xff},
    color.RGBA{0x00, 0x66, 0x00, 0xff},
    color.RGBA{0x00, 0x66, 0x33, 0xff},
    color.RGBA{0x00, 0x66, 0x66, 0xff},
    color.RGBA{0x00, 0x66, 0x99, 0xff},
    color.RGBA{0x00, 0x66, 0xcc, 0xff},
    color.RGBA{0x00, 0x66, 0xff, 0xff},
    color.RGBA{0x00, 0x99, 0x00, 0xff},
    color.RGBA{0x00, 0x99, 0x33, 0xff},
    color.RGBA{0x00, 0x99, 0x66, 0xff},
    color.RGBA{0x00, 0x99, 0x99, 0xff},
    color.RGBA{0x00, 0x99, 0xcc, 0xff},
    color.RGBA{0x00, 0x99, 0xff, 0xff},
    color.RGBA{0x00, 0xcc, 0x00, 0xff},
    color.RGBA{0x00, 0xcc, 0x33, 0xff},
    color.RGBA{0x00, 0xcc, 0x66, 0xff},
```

```
        color.RGBA{0x00, 0xcc, 0x99, 0xff},
        color.RGBA{0x00, 0xcc, 0xcc, 0xff},
        color.RGBA{0x00, 0xcc, 0xff, 0xff},
        color.RGBA{0x00, 0xff, 0x00, 0xff},
        color.RGBA{0x00, 0xff, 0x33, 0xff},
        color.RGBA{0x00, 0xff, 0x66, 0xff},
        color.RGBA{0x00, 0xff, 0x99, 0xff},
        color.RGBA{0x00, 0xff, 0xcc, 0xff},
        color.RGBA{0x00, 0xff, 0xff, 0xff},
        color.RGBA{0x33, 0x00, 0x00, 0xff},
        color.RGBA{0x33, 0x00, 0x33, 0xff},
        color.RGBA{0x33, 0x00, 0x66, 0xff},
        color.RGBA{0x33, 0x00, 0x99, 0xff},
        color.RGBA{0x33, 0x00, 0xcc, 0xff},
        color.RGBA{0x33, 0x00, 0xff, 0xff},
        color.RGBA{0x33, 0x33, 0x00, 0xff},
        color.RGBA{0x33, 0x33, 0x33, 0xff},
        color.RGBA{0x33, 0x33, 0x66, 0xff},
        color.RGBA{0x33, 0x33, 0x99, 0xff},
        color.RGBA{0x33, 0x33, 0xcc, 0xff},
        color.RGBA{0x33, 0x33, 0xff, 0xff},
        color.RGBA{0x33, 0x66, 0x00, 0xff},
        color.RGBA{0x33, 0x66, 0x33, 0xff},
        color.RGBA{0x33, 0x66, 0x66, 0xff},
        color.RGBA{0x33, 0x66, 0x99, 0xff},
        color.RGBA{0x33, 0x66, 0xcc, 0xff},
        color.RGBA{0x33, 0x66, 0xff, 0xff},
        color.RGBA{0x33, 0x99, 0x00, 0xff},
        color.RGBA{0x33, 0x99, 0x33, 0xff},
        color.RGBA{0x33, 0x99, 0x66, 0xff},
        color.RGBA{0x33, 0x99, 0x99, 0xff},
        color.RGBA{0x33, 0x99, 0xcc, 0xff},
        color.RGBA{0x33, 0x99, 0xff, 0xff},
        color.RGBA{0x33, 0xcc, 0x00, 0xff},
        color.RGBA{0x33, 0xcc, 0x33, 0xff},
        color.RGBA{0x33, 0xcc, 0x66, 0xff},
        color.RGBA{0x33, 0xcc, 0x99, 0xff},
        color.RGBA{0x33, 0xcc, 0xcc, 0xff},
        color.RGBA{0x33, 0xcc, 0xff, 0xff},
        color.RGBA{0x33, 0xff, 0x00, 0xff},
        color.RGBA{0x33, 0xff, 0x33, 0xff},
        color.RGBA{0x33, 0xff, 0x66, 0xff},
        color.RGBA{0x33, 0xff, 0x99, 0xff},
        color.RGBA{0x33, 0xff, 0xcc, 0xff},
        color.RGBA{0x33, 0xff, 0xff, 0xff},
        color.RGBA{0x66, 0x00, 0x00, 0xff},
        color.RGBA{0x66, 0x00, 0x33, 0xff},
        color.RGBA{0x66, 0x00, 0x66, 0xff},
        color.RGBA{0x66, 0x00, 0x99, 0xff},
        color.RGBA{0x66, 0x00, 0xcc, 0xff},
        color.RGBA{0x66, 0x00, 0xff, 0xff},
        color.RGBA{0x66, 0x33, 0x00, 0xff},
        color.RGBA{0x66, 0x33, 0x33, 0xff},
```

```
        color.RGBA{0x66, 0x33, 0x66, 0xff},
        color.RGBA{0x66, 0x33, 0x99, 0xff},
        color.RGBA{0x66, 0x33, 0xcc, 0xff},
        color.RGBA{0x66, 0x33, 0xff, 0xff},
        color.RGBA{0x66, 0x66, 0x00, 0xff},
        color.RGBA{0x66, 0x66, 0x33, 0xff},
        color.RGBA{0x66, 0x66, 0x66, 0xff},
        color.RGBA{0x66, 0x66, 0x99, 0xff},
        color.RGBA{0x66, 0x66, 0xcc, 0xff},
        color.RGBA{0x66, 0x66, 0xff, 0xff},
        color.RGBA{0x66, 0x99, 0x00, 0xff},
        color.RGBA{0x66, 0x99, 0x33, 0xff},
        color.RGBA{0x66, 0x99, 0x66, 0xff},
        color.RGBA{0x66, 0x99, 0x99, 0xff},
        color.RGBA{0x66, 0x99, 0xcc, 0xff},
        color.RGBA{0x66, 0x99, 0xff, 0xff},
        color.RGBA{0x66, 0xcc, 0x00, 0xff},
        color.RGBA{0x66, 0xcc, 0x33, 0xff},
        color.RGBA{0x66, 0xcc, 0x66, 0xff},
        color.RGBA{0x66, 0xcc, 0x99, 0xff},
        color.RGBA{0x66, 0xcc, 0xcc, 0xff},
        color.RGBA{0x66, 0xcc, 0xff, 0xff},
        color.RGBA{0x66, 0xff, 0x00, 0xff},
        color.RGBA{0x66, 0xff, 0x33, 0xff},
        color.RGBA{0x66, 0xff, 0x66, 0xff},
        color.RGBA{0x66, 0xff, 0x99, 0xff},
        color.RGBA{0x66, 0xff, 0xcc, 0xff},
        color.RGBA{0x66, 0xff, 0xff, 0xff},
        color.RGBA{0x99, 0x00, 0x00, 0xff},
        color.RGBA{0x99, 0x00, 0x33, 0xff},
        color.RGBA{0x99, 0x00, 0x66, 0xff},
        color.RGBA{0x99, 0x00, 0x99, 0xff},
        color.RGBA{0x99, 0x00, 0xcc, 0xff},
        color.RGBA{0x99, 0x00, 0xff, 0xff},
        color.RGBA{0x99, 0x33, 0x00, 0xff},
        color.RGBA{0x99, 0x33, 0x33, 0xff},
        color.RGBA{0x99, 0x33, 0x66, 0xff},
        color.RGBA{0x99, 0x33, 0x99, 0xff},
        color.RGBA{0x99, 0x33, 0xcc, 0xff},
        color.RGBA{0x99, 0x33, 0xff, 0xff},
        color.RGBA{0x99, 0x66, 0x00, 0xff},
        color.RGBA{0x99, 0x66, 0x33, 0xff},
        color.RGBA{0x99, 0x66, 0x66, 0xff},
        color.RGBA{0x99, 0x66, 0x99, 0xff},
        color.RGBA{0x99, 0x66, 0xcc, 0xff},
        color.RGBA{0x99, 0x66, 0xff, 0xff},
        color.RGBA{0x99, 0x99, 0x00, 0xff},
        color.RGBA{0x99, 0x99, 0x33, 0xff},
        color.RGBA{0x99, 0x99, 0x66, 0xff},
        color.RGBA{0x99, 0x99, 0x99, 0xff},
        color.RGBA{0x99, 0x99, 0xcc, 0xff},
        color.RGBA{0x99, 0x99, 0xff, 0xff},
        color.RGBA{0x99, 0xcc, 0x00, 0xff},
```

```
        color.RGBA{0x99, 0xcc, 0x33, 0xff},
        color.RGBA{0x99, 0xcc, 0x66, 0xff},
        color.RGBA{0x99, 0xcc, 0x99, 0xff},
        color.RGBA{0x99, 0xcc, 0xcc, 0xff},
        color.RGBA{0x99, 0xcc, 0xff, 0xff},
        color.RGBA{0x99, 0xff, 0x00, 0xff},
        color.RGBA{0x99, 0xff, 0x33, 0xff},
        color.RGBA{0x99, 0xff, 0x66, 0xff},
        color.RGBA{0x99, 0xff, 0x99, 0xff},
        color.RGBA{0x99, 0xff, 0xcc, 0xff},
        color.RGBA{0x99, 0xff, 0xff, 0xff},
        color.RGBA{0xcc, 0x00, 0x00, 0xff},
        color.RGBA{0xcc, 0x00, 0x33, 0xff},
        color.RGBA{0xcc, 0x00, 0x66, 0xff},
        color.RGBA{0xcc, 0x00, 0x99, 0xff},
        color.RGBA{0xcc, 0x00, 0xcc, 0xff},
        color.RGBA{0xcc, 0x00, 0xff, 0xff},
        color.RGBA{0xcc, 0x33, 0x00, 0xff},
        color.RGBA{0xcc, 0x33, 0x33, 0xff},
        color.RGBA{0xcc, 0x33, 0x66, 0xff},
        color.RGBA{0xcc, 0x33, 0x99, 0xff},
        color.RGBA{0xcc, 0x33, 0xcc, 0xff},
        color.RGBA{0xcc, 0x33, 0xff, 0xff},
        color.RGBA{0xcc, 0x66, 0x00, 0xff},
        color.RGBA{0xcc, 0x66, 0x33, 0xff},
        color.RGBA{0xcc, 0x66, 0x66, 0xff},
        color.RGBA{0xcc, 0x66, 0x99, 0xff},
        color.RGBA{0xcc, 0x66, 0xcc, 0xff},
        color.RGBA{0xcc, 0x66, 0xff, 0xff},
        color.RGBA{0xcc, 0x99, 0x00, 0xff},
        color.RGBA{0xcc, 0x99, 0x33, 0xff},
        color.RGBA{0xcc, 0x99, 0x66, 0xff},
        color.RGBA{0xcc, 0x99, 0x99, 0xff},
        color.RGBA{0xcc, 0x99, 0xcc, 0xff},
        color.RGBA{0xcc, 0x99, 0xff, 0xff},
        color.RGBA{0xcc, 0xcc, 0x00, 0xff},
        color.RGBA{0xcc, 0xcc, 0x33, 0xff},
        color.RGBA{0xcc, 0xcc, 0x66, 0xff},
        color.RGBA{0xcc, 0xcc, 0x99, 0xff},
        color.RGBA{0xcc, 0xcc, 0xcc, 0xff},
        color.RGBA{0xcc, 0xcc, 0xff, 0xff},
        color.RGBA{0xcc, 0xff, 0x00, 0xff},
        color.RGBA{0xcc, 0xff, 0x33, 0xff},
        color.RGBA{0xcc, 0xff, 0x66, 0xff},
        color.RGBA{0xcc, 0xff, 0x99, 0xff},
        color.RGBA{0xcc, 0xff, 0xcc, 0xff},
        color.RGBA{0xcc, 0xff, 0xff, 0xff},
        color.RGBA{0xff, 0x00, 0x00, 0xff},
        color.RGBA{0xff, 0x00, 0x33, 0xff},
        color.RGBA{0xff, 0x00, 0x66, 0xff},
        color.RGBA{0xff, 0x00, 0x99, 0xff},
        color.RGBA{0xff, 0x00, 0xcc, 0xff},
        color.RGBA{0xff, 0x00, 0xff, 0xff},
```

```
        color.RGBA{0xff, 0x33, 0x00, 0xff},
        color.RGBA{0xff, 0x33, 0x33, 0xff},
        color.RGBA{0xff, 0x33, 0x66, 0xff},
        color.RGBA{0xff, 0x33, 0x99, 0xff},
        color.RGBA{0xff, 0x33, 0xcc, 0xff},
        color.RGBA{0xff, 0x33, 0xff, 0xff},
        color.RGBA{0xff, 0x66, 0x00, 0xff},
        color.RGBA{0xff, 0x66, 0x33, 0xff},
        color.RGBA{0xff, 0x66, 0x66, 0xff},
        color.RGBA{0xff, 0x66, 0x99, 0xff},
        color.RGBA{0xff, 0x66, 0xcc, 0xff},
        color.RGBA{0xff, 0x66, 0xff, 0xff},
        color.RGBA{0xff, 0x99, 0x00, 0xff},
        color.RGBA{0xff, 0x99, 0x33, 0xff},
        color.RGBA{0xff, 0x99, 0x66, 0xff},
        color.RGBA{0xff, 0x99, 0x99, 0xff},
        color.RGBA{0xff, 0x99, 0xcc, 0xff},
        color.RGBA{0xff, 0x99, 0xff, 0xff},
        color.RGBA{0xff, 0xcc, 0x00, 0xff},
        color.RGBA{0xff, 0xcc, 0x33, 0xff},
        color.RGBA{0xff, 0xcc, 0x66, 0xff},
        color.RGBA{0xff, 0xcc, 0x99, 0xff},
        color.RGBA{0xff, 0xcc, 0xcc, 0xff},
        color.RGBA{0xff, 0xcc, 0xff, 0xff},
        color.RGBA{0xff, 0xff, 0x00, 0xff},
        color.RGBA{0xff, 0xff, 0x33, 0xff},
        color.RGBA{0xff, 0xff, 0x66, 0xff},
        color.RGBA{0xff, 0xff, 0x99, 0xff},
        color.RGBA{0xff, 0xff, 0xcc, 0xff},
        color.RGBA{0xff, 0xff, 0xff, 0xff},
}
```

WebSafe是一个216色的调色板，被早期版本的Netscape Navigator（一种浏览器）广泛使用，它也被称为Netcape色彩立方。细节参见 http://en.wikipedia.org/wiki/Web_colors#Web-safe_colors

package draw

```
import "image/draw"
```

draw包提供了图像合成函数。参见：http://golang.org/doc/articles/image_draw.html

Index

type Quantizer

```
type Quantizer interface {
    // Quantize方法向p中最多增加cap(p)-len(p)个色彩并返回修正后的调色板
    // 返回值更适合（可以更少失真的）将m转换为一个调色板类型的图像
    Quantize(p color.Palette, m image.Image) color.Palette
}
```

Quantizer接口为一个图像生成调色板。

type Image

```
type Image interface {
    image.Image
    Set(x, y int, c color.Color)
}
```

Image接口比image.Image接口多了Set方法，该方法用于修改单个像素的色彩。

type Drawer

```
type Drawer interface {
    // 对齐图像dst的r.Min和src的sp点，然后将src绘制到dst上的结果来替换矩形
    Draw(dst Image, r image.Rectangle, src image.Image, sp image.Po
}
```

Drawer包含Draw方法。

```
var FloydSteinberg Drawer = floydSteinberg{}
```

FloydSteinberg是采用Src操作并实现了Floyd-Steinberg误差扩散算法的Drawer。

type Op

```
type Op int
```

Op是Porter-Duff合成的操作符。

```
const (
    // 源图像透过遮罩后，覆盖在目标图像上（类似图层）
    Over Op  = iota
    // 源图像透过遮罩后，替换掉目标图像
    Src
)
```

func (Op) Draw

```
func (op Op) Draw(dst Image, r image.Rectangle, src image.Image, sp
```

Draw方法通过使用Op参数调用包的Draw函数实现了Drawer接口。

func Draw

```
func Draw(dst Image, r image.Rectangle, src image.Image, sp image.P
```

Draw函数使用nil的mask参数调用DrawMask函数。

func DrawMask

```
func DrawMask(dst Image, r image.Rectangle, src image.Image, sp ima
```

对齐目标图像dst的矩形r左上角、源图像src的sp点、遮罩mask的mp点，根据op修改dst的r矩形区域内的内容，mask如果为nil则视为完全透明。

package gif

```
import "image/gif"
```

gif包实现了gif文件的编码器和解码器。gif格式参见：http://www.w3.org/Graphics/GIF/spec-gif89a.txt

Index

type Options

```
type Options struct {
    // NumColors是图像中的最多颜色数，范围[1, 256]
    NumColors int
    // Quantizer用于生成NumColors大小的调色板，为nil时默认使用palette.Pl
    Quantizer draw.Quantizer
    // Drawer用于将图像写入期望的调色板格式的图像，为nil时会使用draw.FloydS
    Drawer draw.Drawer
}
```

Options是编码参数。

func Decode

```
func Decode(r io.Reader) (image.Image, error)
```

从文件流解码并返回GIF文件中的第一幅图像。

func DecodeConfig

```
func DecodeConfig(r io.Reader) (image.Config, error)
```

返回GIF图像的色彩模型和尺寸；函数不会解码整个图像文件。

func Encode

```
func Encode(w io.Writer, m image.Image, o *Options) error
```

函数将图像以GIF格式写入w中。

type GIF

```
type GIF struct {
    Image     []*image.Paletted // 连续的图像
    Delay     []int             // 每一帧延迟时间，单位是0.01s
    LoopCount int               // 总的循环时间
}
```

GIF类型代表可能保存在GIF文件里的多幅图像。

func DecodeAll

```
func DecodeAll(r io.Reader) (*GIF, error)
```

函数从r中读取一个GIF格式文件；返回值中包含了连续的图帧和时间信息。

func EncodeAll

```
func EncodeAll(w io.Writer, g *GIF) error
```

函数将g中所有的图像按指定的每帧延迟和累计循环时间写入w中。

package jpeg

```
import "image/jpeg"
```

jpeg包实现了jpeg格式图像的编解码。JPEG格式参见 http://www.w3.org/Graphics/JPEG/itu-t81.pdf

Index

Constants

```
const DefaultQuality = 75
```

DefaultQuality是默认的编码质量参数。

type Reader

```
type Reader interface {
    io.Reader
    ReadByte() (c byte, err error)
}
```

如果提供的io.Reader接口没有ReadByte方法，Decode函数会为该接口附加一个缓冲。

type FormatError

```
type FormatError string
```

当输入流不是合法的jpeg格式图像时，就会返回FormatError类型的错误。

func (FormatError) Error

```
func (e FormatError) Error() string
```

type UnsupportedError

```
type UnsupportedError string
```

当输入流使用了合法但尚不支持的jpeg特性的时候，就会返回UnsupportedError类型的错误。

func (UnsupportedError) Error

```
func (e UnsupportedError) Error() string
```

type Options

```
type Options struct {
    Quality int
}
```

Options是编码质量参数。取值范围[1,100]，越大图像编码质量越高。

func Decode

```
func Decode(r io.Reader) (image.Image, error)
```

从r读取一幅jpeg格式的图像并解码返回该图像。

func DecodeConfig

```
func DecodeConfig(r io.Reader) (image.Config, error)
```

返回JPEG图像的色彩模型和尺寸；函数不会解码整个图像。

func Encode

```
func Encode(w io.Writer, m image.Image, o *Options) error
```

Encode函数将采用JPEG 4:2:0基线格式和指定的编码质量将图像写入w。如果o为nil将使用DefaultQuality。

package png

```
import "image/png"
```

png包实现了PNG图像的编解码。PNG格式参见：http://www.w3.org/TR/PNG/

Index

type FormatError

```
type FormatError string
```

当输入流不是合法的png格式图像时，就会返回FormatError类型的错误。

func (FormatError) Error

```
func (e FormatError) Error() string
```

type UnsupportedError

```
type UnsupportedError string
```

当输入流使用了合法但尚不支持的png特性的时候，就会返回UnsupportedError类型的错误。

func (UnsupportedError) Error

```
func (e UnsupportedError) Error() string
```

func Decode

```
func Decode(r io.Reader) (image.Image, error)
```

从r读取一幅png格式的图像并解码返回该图像。图像的具体类型要看png文件的内容而定。

func DecodeConfig

```
func DecodeConfig(r io.Reader) (image.Config, error)
```

返回PNG图像的色彩模型和尺寸；函数不会解码整个图像。

func Encode

```
func Encode(w io.Writer, m image.Image) error
```

将图像m以PNG格式写入w。任意图像类型都可以被编码，但image.NRGBA以外格式的图像可能会在编码时丢失一些图像信息。

package index

package suffixarray

```
import "index/suffixarray"
```

suffixarrayb包通过使用内存中的后缀树实现了对数级时间消耗的子字符串搜索。

用法举例：

```
// 创建数据的索引
index := suffixarray.New(data)
// 查找切片s
offsets1 := index.Lookup(s, -1) // 返回data中所有s出现的位置
offsets2 := index.Lookup(s, 3)  // 返回data中最多3个所有s出现的位置
```

Index

type Index

```
type Index struct {
    // 内含隐藏或非导出字段
}
```

Index类型实现了用于快速子字符串搜索的后缀数组。

func New

```
func New(data []byte) *Index
```

使用给出的[]byte数据生成一个*Index，时间复杂度O(N*log(N))。

func (*Index) Bytes

```
func (x *Index) Bytes() []byte
```

返回创建x时提供的[]byte数据，注意不能修改返回值。

func (*Index) Read

```
func (x *Index) Read(r io.Reader) error
```

从r中读取一个index写入x，x不能为nil。

func (*Index) Write

```
func (x *Index) Write(w io.Writer) error
```

将x中的index写入w中，x不能为nil。

func (*Index) Lookup

```
func (x *Index) Lookup(s []byte, n int) (result []int)
```

返回一个未排序的列表，内为s在被索引为index的切片数据中出现的位置。如果n<0，返回全部匹配；如果n==0或s为空，返回nil；否则n为result的最大长度。时间复杂度O(log(N)*len(s) + len(result))，其中N是被索引的数据的大小。

func (*Index) FindAllIndex

```
func (x *Index) FindAllIndex(r *regexp.Regexp, n int) (result [][]i
```

返回一个正则表达式r的不重叠的匹配的经过排序的列表，一个匹配表示为一对指定了匹配结果的切片的索引（相对于x.Bytes()）。如果n<0，返回全部匹配；如果n==0或匹配失败，返回nil；否则n为result的最大长度。

package io

```
import "io"
```

io包提供了对I/O原语的基本接口。本包的基本任务是包装这些原语已有的实现（如os包里的原语），使之成为共享的公共接口，这些公共接口抽象出了泛用的函数并附加了一些相关的原语的操作。

因为这些接口和原语是对底层实现完全不同的低水平操作的包装，除非得到其它方面的通知，客户端不应假设它们是并发执行安全的。

Index

Variables

```
var EOF = errors.New("EOF")
```

EOF当无法得到更多输入时，Read方法返回EOF。当函数一切正常的到达输入的结束时，就应返回EOF。如果在一个结构化数据流中EOF在不期望的位置出现了，则应返回错误ErrUnexpectedEOF或者其它给出更多细节的错误。

```
var ErrClosedPipe = errors.New("io: read/write on closed pipe")
```

当从一个已关闭的Pipe读取或者写入时，会返回ErrClosedPipe。

```
var ErrNoProgress = errors.New("multiple Read calls return no data
```

某些使用io.Reader接口的客户端如果多次调用Read都不返回数据也不返回错误时，就会返回本错误，一般来说是io.Reader的实现有问题的标志。

```
var ErrShortBuffer = errors.New("short buffer")
```

ErrShortBuffer表示读取操作需要大缓冲，但提供的缓冲不够大。

```
var ErrShortWrite = errors.New("short write")
```

ErrShortWrite表示写入操作写入的数据比提供的少，却没有显式的返回错误。

```
var ErrUnexpectedEOF = errors.New("unexpected EOF")
```

ErrUnexpectedEOF表示在读取一个固定尺寸的块或者数据结构时，在读取未完全时遇到了EOF。

type Reader

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Reader接口用于包装基本的读取方法。

Read方法读取len(p)字节数据写入p。它返回写入的字节数和遇到的任何错误。即使Read方法返回值n < len(p)，本方法在被调用时仍可能使用p的全部长度作为暂存空间。如果有部分可用数据，但不够len(p)字节，Read按惯例会返回可以读取到的数据，而不是等待更多数据。

当Read在读取n > 0个字节后遭遇错误或者到达文件结尾时，会返回读取的字节数。它可能会在该次调用返回一个非nil的错误，或者在下一次调用时返回0和该错误。一个常见的例子，Reader接口会在输入流的结尾返回非0的字节数，返回值err == EOF或err == nil。但不管怎样，下一次Read调用必然返回(0, EOF)。调用者应该总是先处理读取的n > 0字节再处理错误值。这么做可以正确的处理发生在读取部分数据后的I/O错误，也能正确处理EOF事件。

如果Read的某个实现返回0字节数和nil错误值，表示被阻碍；调用者应该将这种情况视为未进行操作。

type Writer

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Writer接口用于包装基本的写入方法。

Write方法len(p) 字节数据从p写入底层的数据流。它会返回写入的字节数(0 <= n <= len(p))和遇到的任何导致写入提取结束的错误。Write必须返回非nil的错误，如果它返回的 n < len(p)。Write不能修改切片p中的数据，即使临时修改也不行。

type Closer

```
type Closer interface {
    Close() error
}
```

Closer接口用于包装基本的关闭方法。

在第一次调用之后再次被调用时，Close方法的的行为是未定义的。某些实现可能会说明他们自己的行为。

type Seeker

```
type Seeker interface {
    Seek(offset int64, whence int) (int64, error)
}
```

Seeker接口用于包装基本的移位方法。

Seek方法设定下一次读写的位置：偏移量为offset，校准点由whence确定：0表示相对于文件起始；1表示相对于当前位置；2表示相对于文件结尾。Seek方法返回新的位置以及可能遇到的错误。

移动到一个绝对偏移量为负数的位置会导致错误。移动到任何偏移量为正数的位置都是合法的，但其下一次I/O操作的具体行为则要看底层的实现。

type ReadCloser

```
type ReadCloser interface {
    Reader
    Closer
}
```

ReadCloser接口聚合了基本的读取和关闭操作。

type ReadSeeker

```
type ReadSeeker interface {
    Reader
    Seeker
}
```

ReadSeeker接口聚合了基本的读取和移位操作。

type WriteCloser

```
type WriteCloser interface {
    Writer
    Closer
}
```

WriteCloser接口聚合了基本的写入和关闭操作。

type WriteSeeker

```
type WriteSeeker interface {
    Writer
    Seeker
}
```

WriteSeeker接口聚合了基本的写入和移位操作。

type ReadWriter

```
type ReadWriter interface {
    Reader
    Writer
}
```

ReadWriter接口聚合了基本的读写操作。

type ReadWriteCloser

```
type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

ReadWriteCloser接口聚合了基本的读写和关闭操作。

type ReadWriteSeeker

```
type ReadWriteSeeker interface {
    Reader
    Writer
    Seeker
}
```

ReadWriteSeeker接口聚合了基本的读写和移位操作。

type ReaderAt

```
type ReaderAt interface {
    ReadAt(p []byte, off int64) (n int, err error)
}
```

ReaderAt接口包装了基本的ReadAt方法。

ReadAt从底层输入流的偏移量off位置读取len(p)字节数据写入p， 它返回读取的字节数(0 <= n <= len(p))和遇到的任何错误。当ReadAt方法返回值n < len(p)时，它会返回一个非nil的错误来说明为啥没有读取更多的字节。在这方面，ReadAt是比Read要严格的。即使ReadAt方法返回值 n < len(p)，它在被调用时仍可能使用p的全部长度作为暂存空间。如果有部分可用数据，但不够len(p)字节，ReadAt会阻塞直到获取len(p)个字节数据或者遇到错误。在这方面，ReadAt和Read是不同的。如果ReadAt返回时到达输入流的结尾，而返回值n == len(p)，其返回值err既可以是EOF也可以是nil。

如果ReadAt是从某个有偏移量的底层输入流（的Reader包装）读取，ReadAt方法既不应影响底层的偏移量，也不应被底层的偏移量影响。

ReadAt方法的调用者可以对同一输入流执行并行的ReadAt调用。

type WriterAt

```
type WriterAt interface {
    WriteAt(p []byte, off int64) (n int, err error)
}
```

WriterAt接口包装了基本的WriteAt方法。

WriteAt将p全部len(p)字节数据写入底层数据流的偏移量off位置。它返回写入的字节数(0 <= n <= len(p))和遇到的任何导致写入提前中止的错误。当其返回值n < len(p)时，WriteAt必须放哪会一个非nil的错误。

如果WriteAt写入的对象是某个有偏移量的底层输出流（的Writer包装），WriteAt方法既不应影响底层的偏移量，也不应被底层的偏移量影响。

ReadAt方法的调用者可以对同一输入流执行并行的WriteAt调用。（前提是写入范围不重叠）

type ByteReader

```
type ByteReader interface {
    ReadByte() (c byte, err error)
}
```

ByteReader是基本的ReadByte方法的包装。

ReadByte读取输入中的单个字节并返回。如果没有字节可读取，会返回错误。

type ByteScanner

```
type ByteScanner interface {
    ByteReader
    UnreadByte() error
}
```

ByteScanner接口在基本的ReadByte方法之外还添加了UnreadByte方法。

UnreadByte方法让下一次调用ReadByte时返回之前调用ReadByte时返回的同一个字节。连续调用两次UnreadByte方法而中间没有调用ReadByte时，可能会导致错误。

type RuneReader

```
type RuneReader interface {
    ReadRune() (r rune, size int, err error)
}
```

RuneReader是基本的ReadRune方法的包装。

ReadRune读取单个utf-8编码的字符，返回该字符和它的字节长度。如果没有有效的字符，会返回错误。

type RuneScanner

```
type RuneScanner interface {
    RuneReader
    UnreadRune() error
}
```

RuneScanner接口在基本的ReadRune方法之外还添加了UnreadRune方法。

UnreadRune方法让下一次调用ReadRune时返回之前调用ReadRune时返回的同一个utf-8字符。连续调用两次UnreadRune方法而中间没有调用ReadRune时，可能会导致错误。

type ByteWriter

```
type ByteWriter interface {
    WriteByte(c byte) error
}
```

ByteWriter是基本的WriteByte方法的包装。

type ReaderFrom

```
type ReaderFrom interface {
    ReadFrom(r Reader) (n int64, err error)
}
```

ReaderFrom接口包装了基本的ReadFrom方法。

ReadFrom方法从r读取数据直到EOF或者遇到错误。返回值n是读取的字节数，执行时遇到的错误（EOF除外）也会被返回。

type WriterTo

```
type WriterTo interface {
    WriteTo(w Writer) (n int64, err error)
}
```

WriterTo接口包装了基本的WriteTo方法。

WriteTo方法将数据写入w直到没有数据可以写入或者遇到错误。返回值n是写入的字节数，执行时遇到的任何错误也会被返回。

type LimitedReader

```
type LimitedReader struct {
    R   Reader // 底层Reader接口
    N   int64  // 剩余可读取字节数
}
```

LimitedReader从R中读取数据，但限制可以读取的数据的量为最多N字节，每次调用Read方法都会更新N以标记剩余可以读取的字节数。

func LimitReader

```
func LimitReader(r Reader, n int64) Reader
```

返回一个Reader，它从r中读取n个字节后以EOF停止。返回值接口的底层为*LimitedReader类型。

func (*LimitedReader) Read

```
func (l *LimitedReader) Read(p []byte) (n int, err error)
```

type SectionReader

```
type SectionReader struct {
    // 内含隐藏或非导出字段
}
```

SectionReader实现了对底层满足ReadAt接口的输入流某个片段的Read、ReadAt、Seek方法。

func NewSectionReader

```
func NewSectionReader(r ReaderAt, off int64, n int64) *SectionReade
```

返回一个从r中的偏移量off处为起始，读取n个字节后以EOF停止的SectionReader。

func (*SectionReader) Size

```
func (s *SectionReader) Size() int64
```

Size返回该片段的字节数。

func (*SectionReader) Read

```
func (s *SectionReader) Read(p []byte) (n int, err error)
```

func (*SectionReader) ReadAt

```
func (s *SectionReader) ReadAt(p []byte, off int64) (n int, err err
```

func (*SectionReader) Seek

```
func (s *SectionReader) Seek(offset int64, whence int) (int64, erro
```

type PipeReader

```
type PipeReader struct {
    // 内含隐藏或非导出字段
}
```

PipeReader是一个管道的读取端。

func Pipe

```
func Pipe() (*PipeReader, *PipeWriter)
```

Pipe创建一个同步的内存中的管道。它可以用于连接期望io.Reader的代码和期望io.Writer的代码。一端的读取对应另一端的写入，直接在两端拷贝数据，没有内部缓冲。可以安全的并行调用Read和Write或者Read/Write与Close方法。Close方法会在最后一次阻塞中的I/O操作结束后完成。并行调用Read或并行调用Write也是安全的：每一个独立的调用会依次进行。

func (*PipeReader) Read

```
func (r *PipeReader) Read(data []byte) (n int, err error)
```

Read实现了标准Reader接口：它从管道中读取数据，会阻塞直到写入端开始写入或写入端被关闭。

func (*PipeReader) Close

```
func (r *PipeReader) Close() error
```

Close关闭读取器；关闭后如果对管道的写入端进行写入操作，就会返回(0, ErrClosedPip)。

func (*PipeReader) CloseWithError

```
func (r *PipeReader) CloseWithError(err error) error
```

CloseWithError类似Close方法，但将调用Write时返回的错误改为err。

type PipeWriter

```
type PipeWriter struct {
    // 内含隐藏或非导出字段
}
```

PipeWriter是一个管道的写入端。

func (*PipeWriter) Write

```
func (w *PipeWriter) Write(data []byte) (n int, err error)
```

Write实现了标准Writer接口：它将数据写入到管道中，会阻塞直到读取器读完所有的数据或读取端被关闭。

func (*PipeWriter) Close

```
func (w *PipeWriter) Close() error
```

Close关闭写入器；关闭后如果对管道的读取端进行读取操作，就会返回(0, EOF)。

func (*PipeWriter) CloseWithError

```
func (w *PipeWriter) CloseWithError(err error) error
```

CloseWithError类似Close方法，但将调用Read时返回的错误改为err。

func TeeReader

```
func TeeReader(r Reader, w Writer) Reader
```

TeeReader返回一个将其从r读取的数据写入w的Reader接口。所有通过该接口对r的读取都会执行对应的对w的写入。没有内部的缓冲：写入必须在读取完成前完成。写入时遇到的任何错误都会作为读取错误返回。

func MultiReader

```
func MultiReader(readers ...Reader) Reader
```

MultiReader返回一个将提供的Reader在逻辑上串联起来的Reader接口。他们依次被读取。当所有的输入流都读取完毕，Read才会返回EOF。如果readers中任一个返回了非nil非EOF的错误，Read方法会返回该错误。

func MultiWriter

```
func MultiWriter(writers ...Writer) Writer
```

MultiWriter创建一个Writer接口，会将提供给其的数据写入所有创建时提供的Writer接口。

func Copy

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

将src的数据拷贝到dst，直到在src上到达EOF或发生错误。返回拷贝的字节数和遇到的第一个错误。

对成功的调用，返回值err为nil而非EOF，因为Copy定义为从src读取直到EOF，它不会将读取到EOF视为应报告的错误。如果src实现了WriterTo接口，本函数会调用src.WriteTo(dst)进行拷贝；否则如果dst实现了ReaderFrom接口，本函数会调用dst.ReadFrom(src)进行拷贝。

func CopyN

```
func CopyN(dst Writer, src Reader, n int64) (written int64, err err
```

从src拷贝n个字节数据到dst，直到在src上到达EOF或发生错误。返回复制的字节数和遇到的第一个错误。

只有err为nil时，written才会等于n。如果dst实现了ReaderFrom接口，本函数很调用它实现拷贝。

func ReadAtLeast

```
func ReadAtLeast(r Reader, buf []byte, min int) (n int, err error)
```

ReadAtLeast从r至少读取min字节数据填充进buf。函数返回写入的字节数和错误（如果没有读取足够的字节）。只有没有读取到字节时才可能返回EOF；如果读取了有但不够的字节时遇到了EOF，函数会返回ErrUnexpectedEOF。 如果min比buf的长度还大，函数会返回ErrShortBuffer。只有返回值err为nil时，返回值n才会不小于min。

func ReadFull

```
func ReadFull(r Reader, buf []byte) (n int, err error)
```

ReadFull从r精确地读取len(buf)字节数据填充进buf。函数返回写入的字节数和错误（如果没有读取足够的字节）。只有没有读取到字节时才可能返回EOF；如果读取了有但不够的字节时遇到了EOF，函数会返回ErrUnexpectedEOF。 只有返回值err为nil时，返回值n才会等于len(buf)。

func WriteString

```
func WriteString(w Writer, s string) (n int, err error)
```

WriteString函数将字符串s的内容写入w中。如果w已经实现了WriteString方法，函数会直接调用该方法。

package ioutil

```
import "io/ioutil"
```

Package ioutil implements some I/O utility functions.

Index

Variables

```
var Discard io.Writer = devNull(0)
```

Discard是一个io.Writer接口，对它的所有Write调用都会无实际操作的成功返回。

func NopCloser

```
func NopCloser(r io.Reader) io.ReadCloser
```

NopCloser用一个无操作的Close方法包装r返回一个ReadCloser接口。

func ReadAll

```
func ReadAll(r io.Reader) ([]byte, error)
```

ReadAll从r读取数据直到EOF或遇到error，返回读取的数据和遇到的错误。成功的调用返回的err为nil而非EOF。因为本函数定义为读取r直到EOF，它不会将读取返回的EOF视为应报告的错误。

func ReadFile

```
func ReadFile(filename string) ([]byte, error)
```

ReadFile 从filename指定的文件中读取数据并返回文件的内容。成功的调用返回的err为nil而非EOF。因为本函数定义为读取整个文件，它不会将读取返回的EOF视为应报告的错误。

func WriteFile

```
func WriteFile(filename string, data []byte, perm os.FileMode) erro
```

函数向filename指定的文件中写入数据。如果文件不存在将按给出的权限创建文件，否则在写入数据之前清空文件。

func ReadDir

```
func ReadDir(dirname string) ([]os.FileInfo, error)
```

返回dirname指定的目录的目录信息的有序列表。

func TempDir

```
func TempDir(dir, prefix string) (name string, err error)
```

在dir目录里创建一个新的、使用prfix作为前缀的临时文件夹，并返回文件夹的路径。如果dir是空字符串，TempDir使用默认用于临时文件的目录（参见os.TempDir函数）。 不同程序同时调用该函数会创建不同的临时目录，调用本函数的程序有责任在不需要临时文件夹时摧毁它。

func TempFile

```
func TempFile(dir, prefix string) (f *os.File, err error)
```

在dir目录下创建一个新的、使用prefix为前缀的临时文件，以读写模式打开该文件并返回os.File指针。如果dir是空字符串，TempFile使用默认用于临时文件的目录（参见os.TempDir函数）。不同程序同时调用该函数会创建不同的临时文件，调用本函数的程序有责任在不需要临时文件时摧毁它。

package log

```
import "log"
```

log包实现了简单的日志服务。本包定义了Logger类型，该类型提供了一些格式化输出的方法。本包也提供了一个预定义的“标准”Logger，可以通过辅助函数Print[f|ln]、Fatal[f|ln]和Panic[f|ln]访问，比手工创建一个Logger对象更容易使用。Logger会打印每条日志信息的日期、时间，默认输出到标准错误。Fatal系列函数会在写入日志信息后调用os.Exit(1)。Panic系列函数会在写入日志信息后panic。

Index

Examples

- Logger

Constants

```
const (
    // 字位共同控制输出日志信息的细节。不能控制输出的顺序和格式。
    // 在所有项目后会有一个冒号：2009/01/23 01:23:23.123123 /a/b/c/d.go
    Ldate         = 1 << iota     // 日期：2009/01/23
    Ltime                         // 时间：01:23:23
    Lmicroseconds                 // 微秒分辨率：01:23:23.123123（用于
    Llongfile                     // 文件全路径名+行号： /a/b/c/d.go:2
    Lshortfile                    // 文件无路径名+行号：d.go:23（会覆盖
    LstdFlags     = Ldate | Ltime // 标准logger的初始值
)
```

这些选项定义Logger类型如何生成用于每条日志的前缀文本。

type Logger

```
type Logger struct {
    // contains filtered or unexported fields
}
```

Logger类型表示一个活动状态的记录日志的对象，它会生成一行行的输出写入一个io.Writer接口。每一条日志操作会调用一次io.Writer接口的Write方法。Logger类型的对象可以被多个线程安全的同时使用，它会保证对io.Writer接口的顺序访问。

Example

```
var buf bytes.Buffer
logger := log.New(&buf, "logger: ", log.Lshortfile)
logger.Print("Hello, log file!")
fmt.Print(&buf)
```

Output:

```
logger: example_test.go:16: Hello, log file!
```

func New

```
func New(out io.Writer, prefix string, flag int) *Logger
```

New创建一个Logger。参数out设置日志信息写入的目的地。参数prefix会添加到生成的每一条日志前面。参数flag定义日志的属性（时间、文件等等）。

func (*Logger) Flags

```
func (l *Logger) Flags() int
```

Flags返回logger的输出选项。

func (*Logger) SetFlags

```
func (l *Logger) SetFlags(flag int)
```

SetFlags设置logger的输出选项。

func (*Logger) Prefix

```
func (l *Logger) Prefix() string
```

Prefix返回logger的输出前缀。

func (*Logger) SetPrefix

```
func (l *Logger) SetPrefix(prefix string)
```

SSetPrefix设置logger的输出前缀。

func (*Logger) Output

```
func (l *Logger) Output(calldepth int, s string) error
```

Output写入输出一次日志事件。参数s包含在Logger根据选项生成的前缀之后要打印的文本。如果s末尾没有换行会添加换行符。calldepth用于恢复PC，出于一般性而提供，但目前在所有预定义的路径上它的值都为2。

func (*Logger) Printf

```
func (l *Logger) Printf(format string, v ...interface{})
```

Printf调用l.Output将生成的格式化字符串输出到logger，参数用和fmt.Printf相同的方法处理。

func (*Logger) Print

```
func (l *Logger) Print(v ...interface{})
```

Print调用l.Output将生成的格式化字符串输出到logger，参数用和fmt.Print相同的方法处理。

func (*Logger) Println

```
func (l *Logger) Println(v ...interface{})
```

Println调用l.Output将生成的格式化字符串输出到logger，参数用和fmt.Println相同的方法处理。

func (*Logger) Fatalf

```
func (l *Logger) Fatalf(format string, v ...interface{})
```

Fatalf等价于{l.Printf(v...); os.Exit(1)}

func (*Logger) Fatal

```
func (l *Logger) Fatal(v ...interface{})
```

Fatal等价于{l.Print(v...); os.Exit(1)}

func (*Logger) Fatalln

```
func (l *Logger) Fatalln(v ...interface{})
```

Fatalln等价于{l.Println(v...); os.Exit(1)}

func (*Logger) Panicf

```
func (l *Logger) Panicf(format string, v ...interface{})
```

Panicf等价于{l.Printf(v...); panic(...)}

func (*Logger) Panic

```
func (l *Logger) Panic(v ...interface{})
```

Panic等价于{l.Print(v...); panic(...)}

func (*Logger) Panicln

```
func (l *Logger) Panicln(v ...interface{})
```

Panicln等价于{l.Println(v...); panic(...)}

func Flags

```
func Flags() int
```

Flags返回标准logger的输出选项。

func SetFlags

```
func SetFlags(flag int)
```

SetFlags设置标准logger的输出选项。

func Prefix

```
func Prefix() string
```

Prefix返回标准logger的输出前缀。

func SetPrefix

```
func SetPrefix(prefix string)
```

SetPrefix设置标准logger的输出前缀。

func SetOutput

```
func SetOutput(w io.Writer)
```

SetOutput设置标准logger的输出目的地，默认是标准错误输出。

func Printf

```
func Printf(format string, v ...interface{})
```

Printf调用Output将生成的格式化字符串输出到标准logger，参数用和fmt.Printf相同的方法处理。

func Print

```
func Print(v ...interface{})
```

Print调用Output将生成的格式化字符串输出到标准logger，参数用和fmt.Print相同的方法处理。

func Println

```
func Println(v ...interface{})
```

Println调用Output将生成的格式化字符串输出到标准logger，参数用和fmt.Println相同的方法处理。

func Fatalf

```
func Fatalf(format string, v ...interface{})
```

Fatalf等价于{Printf(v...); os.Exit(1)}

func Fatal

```
func Fatal(v ...interface{})
```

Fatal等价于{Print(v...); os.Exit(1)}

func Fatalln

```
func Fatalln(v ...interface{})
```

Fatalln等价于{Println(v...); os.Exit(1)}

func Panicf

```
func Panicf(format string, v ...interface{})
```

Panicf等价于{Printf(v...); panic(...)}

func Panic

```
func Panic(v ...interface{})
```

Panic等价于{Print(v...); panic(...)}

func Panicln

```
func Panicln(v ...interface{})
```

PanicIn等价于{PrintIn(v...); panic(...)}

package syslog

```
import "log.syslog"
```

syslog包提供一个简单的系统日志服务的接口

Index

- Bugs

Bugs

- ☞ 这个包还没有Windows实现.

package math

```
import "math"
```

math包提供了基本的数学常数和数学函数。

Index

Constants

```
const (
    E   = 2.71828182845904523536028747135266249775724709369995957
    Pi  = 3.14159265358979323846264338327950288419716939937510582
    Phi = 1.61803398874989484820458683436563811772030917980576286
    Sqrt2   = 1.41421356237309504880168872420969807856967187537694
    SqrtE   = 1.64872127070012814684865078781416357165377610071014
    SqrtPi  = 1.77245385090551602729816748334114518279754945612238
    SqrtPhi = 1.27201964951406896425242246173749149171560804184009
    Ln2    = 0.693147180559945309417232121458176568075500134360255
    Log2E  = 1 / Ln2
    Ln10   = 2.30258509299404568401799145468436420760110148862877
    Log10E = 1 / Ln10
)
```

数学常数，参见：http://oeis.org/Axxxxxx

```
const (
    MaxFloat32             = 3.40282346638528859811704183484516925
    SmallestNonzeroFloat32 = 1.40129846432481707092372958328991613
    MaxFloat64             = 1.79769313486231570814527423731704356
    SmallestNonzeroFloat64 = 4.94065645841246544176568792868221372
)
```

浮点数的取值极限。Max是该类型所能表示的最大有限值；SmallestNonzero是该类型所能表示的最小非零正数值。

```
const (
    MaxInt8   = 1<<7 - 1
    MinInt8   = -1 << 7
    MaxInt16  = 1<<15 - 1
    MinInt16  = -1 << 15
    MaxInt32  = 1<<31 - 1
    MinInt32  = -1 << 31
    MaxInt64  = 1<<63 - 1
    MinInt64  = -1 << 63
    MaxUint8  = 1<<8 - 1
    MaxUint16 = 1<<16 - 1
    MaxUint32 = 1<<32 - 1
    MaxUint64 = 1<<64 - 1
)
```

整数的取值极限。

func NaN

```
func NaN() float64
```

函数返回一个IEEE 754“这不是一个数字”值。

func IsNaN

```
func IsNaN(f float64) (is bool)
```

报告f是否表示一个NaN（Not A Number）值。

func Inf

```
func Inf(sign int) float64
```

如果sign>=0函数返回正无穷大，否则返回负无穷大。

func IsInf

```
func IsInf(f float64, sign int) bool
```

如果sign > 0，f是正无穷大时返回真；如果sign<0，f是负无穷大时返回真；sign==0则f是两种无穷大时都返回真。

func Float32bits

```
func Float32bits(f float32) uint32
```

函数返回浮点数f的IEEE 754格式二进制表示对应的4字节无符号整数。

func Float32frombits

```
func Float32frombits(b uint32) float32
```

函数返回无符号整数b对应的IEEE 754格式二进制表示的4字节浮点数。

func Float64bits

```
func Float64bits(f float64) uint64
```

函数返回浮点数f的IEEE 754格式二进制表示对应的8字节无符号整数。

func Float64frombits

```
func Float64frombits(b uint64) float64
```

函数返回无符号整数b对应的IEEE 754格式二进制表示的8字节浮点数。

func Signbit

```
func Signbit(x float64) bool
```

如果x是一个负数或者负零，返回真。

func Copysign

```
func Copysign(x, y float64) float64
```

返回拥有x的量值（绝对值）和y的标志位（正负号）的浮点数。

func Ceil

```
func Ceil(x float64) float64
```

返回不小于x的最小整数（的浮点值），特例如下：

```
Ceil(±0) = ±0
Ceil(±Inf) = ±Inf
Ceil(NaN) = NaN
```

func Floor

```
func Floor(x float64) float64
```

返回不大于x的最小整数（的浮点值），特例如下：

```
Floor(±0) = ±0
Floor(±Inf) = ±Inf
Floor(NaN) = NaN
```

func Trunc

```
func Trunc(x float64) float64
```

返回x的整数部分（的浮点值）。特例如下：

```
Trunc(±0) = ±0
Trunc(±Inf) = ±Inf
Trunc(NaN) = NaN
```

func Modf

```
func Modf(f float64) (int float64, frac float64)
```

返回f的整数部分和小数部分，结果的正负号和都x相同；特例如下：

```
Modf(±Inf) = ±Inf, NaN
Modf(NaN) = NaN, NaN
```

func Nextafter

```
func Nextafter(x, y float64) (r float64)
```

参数x到参数y的方向上，下一个可表示的数值；如果x==y将返回x。特例如下：

```
Nextafter(NaN, y) = NaN
Nextafter(x, NaN) = NaN
```

func Abs

```
func Abs(x float64) float64
```

返回x的绝对值；特例如下：

```
Abs(±Inf) = +Inf
Abs(NaN) = NaN
```

func Max

```
func Max(x, y float64) float64
```

返回x和y中最大值，特例如下：

```
Max(x, +Inf) = Max(+Inf, x) = +Inf
Max(x, NaN) = Max(NaN, x) = NaN
Max(+0, ±0) = Max(±0, +0) = +0
Max(-0, -0) = -0
```

func Min

```
func Min(x, y float64) float64
```

返回x和y中最小值，特例如下：

```
Min(x, -Inf) = Min(-Inf, x) = -Inf
Min(x, NaN) = Min(NaN, x) = NaN
Min(-0, ±0) = Min(±0, -0) = -0
```

func Dim

```
func Dim(x, y float64) float64
```

函数返回x-y和0中的最大值，特殊情况：

```
Dim(+Inf, +Inf) = NaN
Dim(-Inf, -Inf) = NaN
Dim(x, NaN) = Dim(NaN, x) = NaN
```

func Mod

```
func Mod(x, y float64) float64
```

取余运算，可以理解为 x-Trunc(x/y)*y，结果的正负号和x相同；特例如下：

```
Mod(±Inf, y) = NaN
Mod(NaN, y) = NaN
Mod(x, 0) = NaN
Mod(x, ±Inf) = x
Mod(x, NaN) = NaN
```

func Remainder

```
func Remainder(x, y float64) float64
```

IEEE 754差数求值，即x减去最接近x/y的整数值（如果有两个整数与x/y距离相同，则取其中的偶数）与y的乘积。特例如下：

```
Remainder(±Inf, y) = NaN
Remainder(NaN, y) = NaN
Remainder(x, 0) = NaN
Remainder(x, ±Inf) = x
Remainder(x, NaN) = NaN
```

func Sqrt

```
func Sqrt(x float64) float64
```

返回x的二次方根，特例如下：

```
Sqrt(+Inf) = +Inf
Sqrt(±0) = ±0
Sqrt(x < 0) = NaN
Sqrt(NaN) = NaN
```

func Cbrt

```
func Cbrt(x float64) float64
```

返回x的三次方根，特例如下：

```
Cbrt(±0) = ±0
Cbrt(±Inf) = ±Inf
Cbrt(NaN) = NaN
```

func Hypot

```
func Hypot(p, q float64) float64
```

返回Sqrt(p*p + q*q)，注意要避免不必要的溢出或下溢。特例如下：

```
Hypot(±Inf, q) = +Inf
Hypot(p, ±Inf) = +Inf
Hypot(NaN, q) = NaN
Hypot(p, NaN) = NaN
```

func Sin

```
func Sin(x float64) float64
```

求正弦。特例如下：

```
Sin(±0) = ±0
Sin(±Inf) = NaN
Sin(NaN) = NaN
```

func Cos

```
func Cos(x float64) float64
```

求余弦。特例如下：

```
Cos(±Inf) = NaN
Cos(NaN) = NaN
```

func Tan

```
func Tan(x float64) float64
```

求正切。特例如下：

```
Tan(±0) = ±0
Tan(±Inf) = NaN
Tan(NaN) = NaN
```

func Sincos

```
func Sincos(x float64) (sin, cos float64)
```

函数返回Sin(x), Cos(x)。特例如下：

```
Sincos(±0) = ±0, 1
Sincos(±Inf) = NaN, NaN
Sincos(NaN) = NaN, NaN
```

func Asin

```
func Asin(x float64) float64
```

求反正弦（x是弧度）。特例如下：

```
Asin(±0) = ±0
Asin(x) = NaN if x < -1 or x > 1
```

func Acos

```
func Acos(x float64) float64
```

求反余弦（x是弧度）。特例如下：

```
Acos(x) = NaN if x < -1 or x > 1
```

func Atan

```
func Atan(x float64) float64
```

求反正切（x是弧度）。特例如下：

```
Atan(±0) = ±0
Atan(±Inf) = ±Pi/2
```

func Atan2

```
func Atan2(y, x float64) float64
```

类似Atan(y/x)，但会根据x，y的正负号确定象限。特例如下（前面的优先）：

```
Atan2(y, NaN) = NaN
Atan2(NaN, x) = NaN
Atan2(+0, x>=0) = +0
Atan2(-0, x>=0) = -0
Atan2(+0, x<=-0) = +Pi
Atan2(-0, x<=-0) = -Pi
Atan2(y>0, 0) = +Pi/2
Atan2(y<0, 0) = -Pi/2
Atan2(+Inf, +Inf) = +Pi/4
Atan2(-Inf, +Inf) = -Pi/4
Atan2(+Inf, -Inf) = 3Pi/4
Atan2(-Inf, -Inf) = -3Pi/4
Atan2(y, +Inf) = 0
Atan2(y>0, -Inf) = +Pi
Atan2(y<0, -Inf) = -Pi
Atan2(+Inf, x) = +Pi/2
Atan2(-Inf, x) = -Pi/2
```

func Sinh

```
func Sinh(x float64) float64
```

求双曲正弦，特例如下：

```
Sinh(±0) = ±0
Sinh(±Inf) = ±Inf
Sinh(NaN) = NaN
```

func Cosh

```
func Cosh(x float64) float64
```

求双曲余弦，特例如下：

```
Cosh(±0) = 1
Cosh(±Inf) = +Inf
Cosh(NaN) = NaN
```

func Tanh

```
func Tanh(x float64) float64
```

求双曲正切，特例如下：

```
Tanh(±0) = ±0
Tanh(±Inf) = ±1
Tanh(NaN) = NaN
```

func Asinh

```
func Asinh(x float64) float64
```

求反双曲正弦，特例如下：

```
Asinh(±0) = ±0
Asinh(±Inf) = ±Inf
Asinh(NaN) = NaN
```

func Acosh

```
func Acosh(x float64) float64
```

求反双曲余弦，特例如下：

```
Acosh(+Inf) = +Inf
Acosh(x) = NaN if x < 1
Acosh(NaN) = NaN
```

func Atanh

```
func Atanh(x float64) float64
```

求反双曲正切，特例如下：

```
Atanh(1) = +Inf
Atanh(±0) = ±0
Atanh(-1) = -Inf
Atanh(x) = NaN if x < -1 or x > 1
Atanh(NaN) = NaN
```

func Log

```
func Log(x float64) float64
```

求自然对数，特例如下：

```
Log(+Inf) = +Inf
Log(0) = -Inf
Log(x < 0) = NaN
Log(NaN) = NaN
```

func Log1p

```
func Log1p(x float64) float64
```

等价于Log(1+x)。但是在x接近0时，本函数更加精确；特例如下：

```
Log1p(+Inf) = +Inf
Log1p(±0) = ±0
Log1p(-1) = -Inf
Log1p(x < -1) = NaN
Log1p(NaN) = NaN
```

func Log2

```
func Log2(x float64) float64
```

求2为底的对数；特例和Log相同。

func Log10

```
func Log10(x float64) float64
```

求10为底的对数；特例和Log相同。

func Logb

```
func Logb(x float64) float64
```

返回x的二进制指数值，可以理解为Trunc(Log2(x))；特例如下：

```
Logb(±Inf) = +Inf
Logb(0) = -Inf
Logb(NaN) = NaN
```

func Ilogb

```
func Ilogb(x float64) int
```

类似Logb，但返回值是整型；特例如下：

```
Ilogb(±Inf) = MaxInt32
Ilogb(0) = MinInt32
Ilogb(NaN) = MaxInt32
```

func Frexp

```
func Frexp(f float64) (frac float64, exp int)
```

返回一个标准化小数frac和2的整型指数exp，满足f == frac * 2**exp，且0.5 <= Abs(frac) < 1；特例如下：

```
Frexp(±0) = ±0, 0
Frexp(±Inf) = ±Inf, 0
Frexp(NaN) = NaN, 0
```

func Ldexp

```
func Ldexp(frac float64, exp int) float64
```

Frexp的反函数，返回 frac * 2**exp。特例如下：

```
Ldexp(±0, exp) = ±0
Ldexp(±Inf, exp) = ±Inf
Ldexp(NaN, exp) = NaN
```

func Exp

```
func Exp(x float64) float64
```

返回E**x；x绝对值很大时可能会溢出为0或者+Inf，x绝对值很小时可能会下溢为1。特例如下：

```
Exp(+Inf) = +Inf
Exp(NaN) = NaN
```

func Expm1

```
func Expm1(x float64) float64
```

等价于Exp(x)-1，但是在x接近零时更精确；x绝对值很大时可能会溢出为-1或+Inf。特例如下：

```
Expm1(+Inf) = +Inf
Expm1(-Inf) = -1
Expm1(NaN) = NaN
```

func Exp2

```
func Exp2(x float64) float64
```

返回2**x；特例和Exp相同。

func Pow

```
func Pow(x, y float64) float64
```

返回x**y；特例如下（前面的优先）：

```
Pow(x, ±0) = 1 for any x
Pow(1, y) = 1 for any y
Pow(x, 1) = x for any x
Pow(NaN, y) = NaN
Pow(x, NaN) = NaN
Pow(±0, y) = ±Inf for y an odd integer < 0
Pow(±0, -Inf) = +Inf
Pow(±0, +Inf) = +0
Pow(±0, y) = +Inf for finite y < 0 and not an odd integer
Pow(±0, y) = ±0 for y an odd integer > 0
Pow(±0, y) = +0 for finite y > 0 and not an odd integer
Pow(-1, ±Inf) = 1
Pow(x, +Inf) = +Inf for |x| > 1
Pow(x, -Inf) = +0 for |x| > 1
Pow(x, +Inf) = +0 for |x| < 1
Pow(x, -Inf) = +Inf for |x| < 1
Pow(+Inf, y) = +Inf for y > 0
Pow(+Inf, y) = +0 for y < 0
Pow(-Inf, y) = Pow(-0, -y)
Pow(x, y) = NaN for finite x < 0 and finite non-integer y
```

func Pow10

```
func Pow10(e int) float64
```

返回10**e；特例如下：

```
Pow10(e) = +Inf for e > 309
Pow10(e) = 0 for e < -324
```

func Gamma

```
func Gamma(x float64) float64
```

伽玛函数（当x为正整数时，值为(x-1)!）。特例如下：

```
Gamma(+Inf) = +Inf
Gamma(+0) = +Inf
Gamma(-0) = -Inf
Gamma(x) = NaN for integer x < 0
Gamma(-Inf) = NaN
Gamma(NaN) = NaN
```

func Lgamma

```
func Lgamma(x float64) (lgamma float64, sign int)
```

返回Gamma(x)的自然对数和正负号。特例如下：

```
Lgamma(+Inf) = +Inf
Lgamma(0) = +Inf
Lgamma(-integer) = +Inf
Lgamma(-Inf) = -Inf
Lgamma(NaN) = NaN
```

func Erf

```
func Erf(x float64) float64
```

误差函数，特例如下：

```
Erf(+Inf) = 1
Erf(-Inf) = -1
Erf(NaN) = NaN
```

func Erfc

```
func Erfc(x float64) float64
```

余补误差函数，特例如下：

```
Erfc(+Inf) = 0
Erfc(-Inf) = 2
Erfc(NaN) = NaN
```

func J0

```
func J0(x float64) float64
```

第一类贝塞尔函数，0阶。特例如下：

```
J0(±Inf) = 0
J0(0) = 1
J0(NaN) = NaN
```

func J1

```
func J1(x float64) float64
```

第一类贝塞尔函数，1阶。特例如下：

```
J1(±Inf) = 0
J1(NaN) = NaN
```

func Jn

```
func Jn(n int, x float64) float64
```

第一类贝塞尔函数，n阶。特例如下：

```
Jn(n, ±Inf) = 0
Jn(n, NaN) = NaN
```

func Y0

```
func Y0(x float64) float64
```

第二类贝塞尔函数，0阶。特例如下：

```
Y0(+Inf) = 0
Y0(0) = -Inf
Y0(x < 0) = NaN
Y0(NaN) = NaN
```

func Y1

```
func Y1(x float64) float64
```

第二类贝塞尔函数，1阶。特例如下：

```
Y1(+Inf) = 0
Y1(0) = -Inf
Y1(x < 0) = NaN
Y1(NaN) = NaN
```

func Yn

```
func Yn(n int, x float64) float64
```

第二类贝塞尔函数，n阶。特例如下：

```
Yn(n, +Inf) = 0
Yn(n > 0, 0) = -Inf
Yn(n < 0, 0) = +Inf if n is odd, -Inf if n is even
Y1(n, x < 0) = NaN
Y1(n, NaN) = NaN
```

package big

```
import "math/big"
```

big包实现了大数字的多精度计算。 支持如下数字类型：

```
- Int    有符号整数
- Rat    有理数
```

方法一般为如下格式：

```
func (z *Int) Op(x, y *Int) *Int    (similar for *Rat)
```

该方法实现了操作z = x Op y：计算并将结果写入z。如果结果是操作数之一，可能会重写该参数（重用其内存）；为了实现链式的计算，计算结果同时会作为返回值。方法返回一个结果而不是让*Int/*Rat调用方法获取另一个操作数。

Index

Examples

Constants

```
const MaxBase = 'z' - 'a' + 10 + 1 // = hexValue('z') + 1
```

MaxBase是字符串转换函数接受的最大进制。

type Word

```
type Word uintptr
```

Word代表一个多精度无符号整数的单个数字。

type Int

```
type Int struct {
    // 内含隐藏或非导出字段
}
```

Int类型代表多精度的整数，零值代表数字0。

func NewInt

```
func NewInt(x int64) *Int
```

创建一个值为x的*Int。

func (*Int) Int64

```
func (x *Int) Int64() int64
```

返回x的int64表示，如果不能用int64表示，结果是未定义的。

func (*Int) Uint64

```
func (x *Int) Uint64() uint64
```

返回x的uint64表示，如果不能用uint64表示，结果是未定义的。

func (*Int) Bytes

```
func (x *Int) Bytes() []byte
```

返回x的绝对值的大端在前的字节切片表示。

func (*Int) String

```
func (x *Int) String() string
```

func (*Int) BitLen

```
func (x *Int) BitLen() int
```

返回x的绝对值的字位数，0的字位数为0。

func (*Int) Bits

```
func (x *Int) Bits() []Word
```

提供了对x的数据不检查而快速的访问，返回构成x的绝对值的小端在前的word切片。该切片与x的底层是同一个数组，本函数用于支持在包外实现缺少的低水平功能，否则不应被使用。

func (*Int) Bit

```
func (x *Int) Bit(i int) uint
```

返回第i个字位的值，即返回(x>>i)&1。i必须不小于0。

func (*Int) SetInt64

```
func (z *Int) SetInt64(x int64) *Int
```

将z设为x并返回z。

func (*Int) SetUint64

```
func (z *Int) SetUint64(x uint64) *Int
```

将z设为x并返回z。

func (*Int) SetBytes

```
func (z *Int) SetBytes(buf []byte) *Int
```

将buf视为一个大端在前的无符号整数，将z设为该值，并返回z。

func (*Int) SetString

```
func (z *Int) SetString(s string, base int) (*Int, bool)
```

将z设为s代表的值（base为基数）。返回z并用一个bool来表明成功与否。如果失败，z的值是不确定的，但返回值为nil。基数必须是0或者2到MaxBase之间的整数。如果基数为0，字符串的前缀决定实际的转换基数："0x"、"0X"表示十六进制；"0b"、"0B"表示二进制；"0"表示八进制；否则为十进制。

Example

```
i := new(big.Int)
i.SetString("644", 8) // octal
fmt.Println(i)
```

Output:

```
420
```

func (*Int) SetBits

```
func (z *Int) SetBits(abs []Word) *Int
```

提供了对z的数据不检查而快速的操作，将abs视为小端在前的word切片并直接赋给z，返回z。会将z的底层设置为abs的同一底层数组，本函数用于支持在包外实现缺少的低水平功能，否则不应被使用。

func (*Int) SetBit

```
func (z *Int) SetBit(x *Int, i int, b uint) *Int
```

将z设为x并设置z的第i位为b，返回z。如b为1，z = x | (1 << i)；如b为0，z = x & ^(1 << i)；否则会panic。

func (*Int) MulRange

```
func (z *Int) MulRange(a, b int64) *Int
```

将z设置为区间[a, b]内所有整数的乘积A(a, b)，并返回z。如果a>b会将z设为1并返回。

func (*Int) Binomial

```
func (z *Int) Binomial(n, k int64) *Int
```

将z设为k次二项式展开第n项的系数C(n, k)，并返回z。

func (*Int) Rand

```
func (z *Int) Rand(rnd *rand.Rand, n *Int) *Int
```

将z设为一个范围在[0, n)的伪随机值，并返回z。

func (*Int) ProbablyPrime

```
func (x *Int) ProbablyPrime(n int) bool
```

对x进行n次Miller-Rabin质数检测。如果方法返回真则x是质数的几率为1-(1/4)**n；否则x不是质数。

func (*Int) Sign

```
func (x *Int) Sign() int
```

返回x的正负号。x<0时返回-1；x>0时返回+1；否则返回0。

func (*Int) Cmp

```
func (x *Int) Cmp(y *Int) (r int)
```

比较x和y的大小。x<y时返回-1；x>y时返回+1；否则返回0。

func (*Int) Not

```
func (z *Int) Not(x *Int) *Int
```

将z设为^x并返回z（按位取反）。

func (*Int) And

```
func (z *Int) And(x, y *Int) *Int
```

将z设为x & y并返回z（按位且）。

func (*Int) Or

```
func (z *Int) Or(x, y *Int) *Int
```

将z设为x | y并返回z（按位或）。

func (*Int) Xor

```
func (z *Int) Xor(x, y *Int) *Int
```

将z设为x ^ y并返回z（按位异或）。

func (*Int) AndNot

```
func (z *Int) AndNot(x, y *Int) *Int
```

将z设为x & (^y)并返回z（按位减）。

func (*Int) Lsh

```
func (z *Int) Lsh(x *Int, n uint) *Int
```

将z设为x << n并返回z（左位移运算）。

func (*Int) Rsh

```
func (z *Int) Rsh(x *Int, n uint) *Int
```

将z设为x >> n并返回z（右位移运算）。

func (*Int) Abs

```
func (z *Int) Abs(x *Int) *Int
```

将z设为|x|并返回z。

func (*Int) Neg

```
func (z *Int) Neg(x *Int) *Int
```

将z设为-x并返回z。

func (*Int) Set

```
func (z *Int) Set(x *Int) *Int
```

将z设为x（生成一个拷贝）并返回z

func (*Int) Add

```
func (z *Int) Add(x, y *Int) *Int
```

将z设为x + y并返回z。

func (*Int) Sub

```
func (z *Int) Sub(x, y *Int) *Int
```

将z设为x - y并返回z。

func (*Int) Mul

```
func (z *Int) Mul(x, y *Int) *Int
```

将z设为x * y并返回z。

func (*Int) Div

```
func (z *Int) Div(x, y *Int) *Int
```

如果y != 0会将z设为x/y并返回z；如果y==0会panic。函数采用欧几里德除法（和Go不同），参见DivMod。

func (*Int) Mod

```
func (z *Int) Mod(x, y *Int) *Int
```

如果y != 0会将z设为x%y并返回z；如果y==0会panic。函数采用欧几里德除法（和Go不同），参见DivMod。

func (*Int) DivMod

```
func (z *Int) DivMod(x, y, m *Int) (*Int, *Int)
```

如果y != 0将z设为x/y，将m设为x%y并返回(z, m)；如果y == 0会panic。采用欧几里德除法（和Go不同）

DivMod方法实现了欧几里德带余除法：

```
q = x div y  满足
m = x - y*q  且 0 <= m < |q|
```

func (*Int) Quo

```
func (z *Int) Quo(x, y *Int) *Int
```

如果y != 0会将z设为x/y并返回z；如果y==0会panic。函数采用截断除法（和Go相同），参见QuoRem。

func (*Int) Rem

```
func (z *Int) Rem(x, y *Int) *Int
```

如果y != 0会将z设为x%y并返回z；如果y==0会panic。函数采用截断除法（和Go相同），参见QuoRem。

func (*Int) QuoRem

```
func (z *Int) QuoRem(x, y, r *Int) (*Int, *Int)
```

如果y != 0将z设为x/y，将r设为x%y并返回(z, r)；如果y == 0会panic。函数采用截断除法（和Go相同）

QuoRem方法实现了截断带余除法：

```
q = x/y      返回值向零的方向截断
r = x - y*q
```

func (*Int) ModInverse

```
func (z *Int) ModInverse(g, p *Int) *Int
```

将z设为g相对p的模逆（即z、g满足(z * g) % p == 1）。返回值z大于0小于p。

func (*Int) Exp

```
func (z *Int) Exp(x, y, m *Int) *Int
```

将z设为x**y mod |m|并返回z；如果y <= 0，返回1；如果m == nil 或 m == 0，z设为x**y。

func (*Int) GCD

```
func (z *Int) GCD(x, y, a, b *Int) *Int
```

将z设为a和b的最大公约数并返回z。a或b为nil时会panic；a和b都>0时设置z为最大公约数；如果任一个<=0方法就会设置z = x = y = 0。如果x和y都不是nil，会将x和y设置为满足a*x + b*y==z。

func (*Int) Format

```
func (x *Int) Format(s fmt.State, ch rune)
```

Format方法实现了fmt.Formatter接口。本方法接受格式'b'（二进制）、'o'（八进制）、'd'（十进制）、'x'（小写十六进制）、'X'（大写十六进制）。

方法支持全套fmt包对整数类型的动作：包括用于符号控制的'+'、'-'、' '；用于十六进制和八进制前导0的'#'；"%#x"和"%#X"会设置前导的"0x"或"0X"；指定最小数字精度；输出字段宽度；空格或'0'的补位；左右对齐。

func (*Int) Scan

```
func (z *Int) Scan(s fmt.ScanState, ch rune) error
```

Scan实现了fmt.Scanner接口，将z设为读取的数字。方法可以接受接受格式'b'（二进制）、'o'（八进制）、'd'（十进制）、'x'（小写十六进制）、'X'（大写十六进制）。

Example

```
// The Scan function is rarely used directly;
// the fmt package recognizes it as an implementation of fmt.Scanne
i := new(big.Int)
_, err := fmt.Sscan("18446744073709551617", i)
if err != nil {
    log.Println("error scanning value:", err)
} else {
    fmt.Println(i)
}
```

Output:

```
18446744073709551617
```

func (*Int) GobEncode

```
func (x *Int) GobEncode() ([]byte, error)
```

本方法实现了gob.GobEncoder接口。

func (*Int) GobDecode

```
func (z *Int) GobDecode(buf []byte) error
```

本方法实现了gob.GobDecoder接口。

func (*Int) MarshalJSON

```
func (z *Int) MarshalJSON() ([]byte, error)
```

本方法实现了json.Marshaler接口。

func (*Int) UnmarshalJSON

```
func (z *Int) UnmarshalJSON(text []byte) error
```

本方法实现了json.Unmarshaler接口。

func (*Int) MarshalText

```
func (z *Int) MarshalText() (text []byte, err error)
```

本方法实现了encoding.TextMarshaler接口。

func (*Int) UnmarshalText

```
func (z *Int) UnmarshalText(text []byte) error
```

本方法实现了encoding.TextUnmarshaler接口。

type Rat

```
type Rat struct {
    // 内含隐藏或非导出字段
}
```

Rat类型代表一个任意精度的有理数（底层采用分数表示），Rat的零值代表数字0。

func NewRat

```
func NewRat(a, b int64) *Rat
```

NewRat函数使用分子a和分母b创建一个Rat。

func (*Rat) Num

```
func (x *Rat) Num() *Int
```

返回x的分子，分子可能<=0。返回的是x分子的指针，因此对返回值的操作可能改变x，反之亦然。x的符号与分子的符号是绑定的。

func (*Rat) Denom

```
func (x *Rat) Denom() *Int
```

返回x的分母，分母总是>0。返回的是x分母的指针，因此对返回值的操作可能改变x，反之亦然。

func (*Rat) Float64

```
func (x *Rat) Float64() (f float64, exact bool)
```

返回最接近x的值的float64值，exact用于说明f是否精确的表示了x。 如果x的量级太大或太小不能被float64类型表示，返回无穷和false；f的符号始终与x的符号一致，即使f==0。

func (*Rat) RatString

```
func (x *Rat) RatString() string
```

返回z的字符串表示，如果分母不等于1，格式为"a/b"；否则格式为"a"。

func (*Rat) FloatString

```
func (x *Rat) FloatString(prec int) string
```

返回z的字符串表示为精度为prec的十进制浮点数，最后一位会进行四舍五入。

func (*Rat) String

```
func (x *Rat) String() string
```

返回z的字符串表示，格式为"a/b"（即使分母等于1）。

func (*Rat) IsInt

```
func (x *Rat) IsInt() bool
```

返回x的分母是否为1（即x为整数）。

func (*Rat) SetInt64

```
func (z *Rat) SetInt64(x int64) *Rat
```

将z设为x，并返回z。

func (*Rat) SetFrac64

```
func (z *Rat) SetFrac64(a, b int64) *Rat
```

将z设为a/b，并返回z。

func (*Rat) SetFloat64

```
func (z *Rat) SetFloat64(f float64) *Rat
```

将z设为f的精确值并返回z。如果f不是有穷数（即f为+Inf、+Inf或NaN）时会返回nil。。

func (*Rat) SetInt

```
func (z *Rat) SetInt(x *Int) *Rat
```

将z设为x（生成一个拷贝）并返回z。

func (*Rat) SetFrac

```
func (z *Rat) SetFrac(a, b *Int) *Rat
```

将z设为a/b，并返回z。

func (*Rat) SetString

```
func (z *Rat) SetString(s string) (*Rat, bool)
```

将z设为字符串代表的值，返回z并用一个bool表明是否成功。字符串s的格式可以是形如"a/b"的分数格式，也可以是浮点数后跟可选的指数的科学计数法格式。如果操作失败，z的值是不确定的，但返回值为nil。

Example

```
r := new(big.Rat)
r.SetString("355/113")
fmt.Println(r.FloatString(3))
```

Output:

```
3.142
```

func (*Rat) Sign

```
func (x *Rat) Sign() int
```

返回x的正负号。如x < 0返回-1；如x > 0返回+1；否则返回0。

func (*Rat) Cmp

```
func (x *Rat) Cmp(y *Rat) int
```

比较x和y的大小。如x < y返回-1；如x > y返回+1；否则返回0。

func (*Rat) Abs

```
func (z *Rat) Abs(x *Rat) *Rat
```

将z设为|x|并返回z。

func (*Rat) Neg

```
func (z *Rat) Neg(x *Rat) *Rat
```

将z设为-x并返回z。

func (*Rat) Inv

```
func (z *Rat) Inv(x *Rat) *Rat
```

将z设为1/x并返回z。

func (*Rat) Set

```
func (z *Rat) Set(x *Rat) *Rat
```

将z设为x（生成一个拷贝）并返回z。

func (*Rat) Add

```
func (z *Rat) Add(x, y *Rat) *Rat
```

将z设为x + y并返回z。

func (*Rat) Sub

```
func (z *Rat) Sub(x, y *Rat) *Rat
```

将z设为x - y并返回z。

func (*Rat) Mul

```
func (z *Rat) Mul(x, y *Rat) *Rat
```

将z设为x * y并返回z。

func (*Rat) Quo

```
func (z *Rat) Quo(x, y *Rat) *Rat
```

如果y != 0会将z设为x/y并返回z；如果y==0会panic。

func (*Rat) Scan

```
func (z *Rat) Scan(s fmt.ScanState, ch rune) error
```

本方法实现了fmt.Scanner接口，将z设为读取到的数字。接受格式'e'、'E'、'f'、'F'、'g'、'G'、'v'；它们都是等价的。

Example

```
// The Scan function is rarely used directly;
// the fmt package recognizes it as an implementation of fmt.Scanne
r := new(big.Rat)
_, err := fmt.Sscan("1.5000", r)
if err != nil {
    log.Println("error scanning value:", err)
} else {
    fmt.Println(r)
}
```

Output:

```
3/2
```

func (*Rat) GobEncode

```
func (x *Rat) GobEncode() ([]byte, error)
```

本方法实现了gob.GobEncoder接口。

func (*Rat) GobDecode

```
func (z *Rat) GobDecode(buf []byte) error
```

本方法实现了gob.GobDecoder接口。

func (*Rat) MarshalText

```
func (r *Rat) MarshalText() (text []byte, err error)
```

本方法实现了encoding.TextMarshaler接口。

func (*Rat) UnmarshalText

```
func (r *Rat) UnmarshalText(text []byte) error
```

本方法实现了encoding.TextUnmarshaler接口。

package cmplx

`import "math/cmplx"`

cmplx包提供了复数的常用常数和常用函数。

Index

func NaN

```
func NaN() complex128
```

返回一个复数的“Not A Number”值。

func IsNaN

```
func IsNaN(x complex128) bool
```

如果x的实部或者虚部是“Not A Number”值，则返回真。

func Inf

```
func Inf() complex128
```

返回一个复数的无穷大，complex(+Inf, +Inf)。

func IsInf

```
func IsInf(x complex128) bool
```

如果x的实部或者虚部是无穷（不管正负），则返回真。

func Abs

```
func Abs(x complex128) float64
```

返回x的绝对值（也被称为模）。

func Phase

```
func Phase(x complex128) float64
```

返回x的相位（也被称为幅角），返回值范围[-Pi, Pi]。

func Polar

```
func Polar(x complex128) (r, θ float64)
```

将直角坐标的复数表示为极坐标(r, θ)。其中r是x的绝对值，θ是x的相位，范围[-Pi, Pi]。

func Rect

```
func Rect(r, θ float64) complex128
```

返回极坐标(r, θ)表示的复数。

func Conj

```
func Conj(x complex128) complex128
```

返回x的共轭复数（实部相等，虚部相反）。

func Sqrt

```
func Sqrt(x complex128) complex128
```

返回x的平方根。返回值的实部不小于0，而虚部的正负号和x的虚部相同。

func Log

```
func Log(x complex128) complex128
```

返回x的自然对数。

func Log10

```
func Log10(x complex128) complex128
```

返回x的常用对数。

func Exp

```
func Exp(x complex128) complex128
```

返回e**x。

func Pow

```
func Pow(x, y complex128) complex128
```

返回x**y；有如下特例：

```
Pow(0, ±0)  returns 1+0i
Pow(0, c)   如果image(c)==0，则当real(c)<0时返回Inf+0i；否则返回Inf+Inf
```

func Sin

```
func Sin(x complex128) complex128
```

求正弦。

func Cos

```
func Cos(x complex128) complex128
```

求余弦。

func Tan

```
func Tan(x complex128) complex128
```

求正切。

func Cot

```
func Cot(x complex128) complex128
```

求余切。

func Asin

```
func Asin(x complex128) complex128
```

求反正弦。

func Acos

```
func Acos(x complex128) complex128
```

求反余弦。

func Atan

```
func Atan(x complex128) complex128
```

求反正切。

func Sinh

```
func Sinh(x complex128) complex128
```

求双曲正弦。

func Cosh

```
func Cosh(x complex128) complex128
```

求双曲余弦。

func Tanh

```
func Tanh(x complex128) complex128
```

求双曲正切。

func Asinh

```
func Asinh(x complex128) complex128
```

求反双曲正弦。

func Acosh

```
func Acosh(x complex128) complex128
```

求反双曲余弦。

func Atanh

```
func Atanh(x complex128) complex128
```

求反双曲正切。

package rand

`import "math/rand"`

rand包实现了伪随机数生成器。

随机数从资源生成。包水平的函数都使用的默认的公共资源。该资源会在程序每次运行时都产生确定的序列。如果需要每次运行产生不同的序列，应使用Seed函数进行初始化。默认资源可以安全的用于多go程并发。

Example

```
rand.Seed(42) // Try changing this number!
answers := []string{
    "It is certain",
    "It is decidedly so",
    "Without a doubt",
    "Yes definitely",
    "You may rely on it",
    "As I see it yes",
    "Most likely",
    "Outlook good",
    "Yes",
    "Signs point to yes",
    "Reply hazy try again",
    "Ask again later",
    "Better not tell you now",
    "Cannot predict now",
    "Concentrate and ask again",
    "Don't count on it",
    "My reply is no",
    "My sources say no",
    "Outlook not so good",
    "Very doubtful",
}
fmt.Println("Magic 8-Ball says:", answers[rand.Intn(len(answers))]
```

Output:

```
Magic 8-Ball says: As I see it yes
```

Example (Rand)

```
// Create and seed the generator.
// Typically a non-fixed seed should be used, such as time.Now().Ur
// Using a fixed seed will produce the same output on every run.
r := rand.New(rand.NewSource(99))
// The tabwriter here helps us generate aligned output.
w := tabwriter.NewWriter(os.Stdout, 1, 1, 1, ' ', 0)
defer w.Flush()
show := func(name string, v1, v2, v3 interface{}) {
    fmt.Fprintf(w, "%s\t%v\t%v\t%v\n", name, v1, v2, v3)
}
// Float32 and Float64 values are in [0, 1).
show("Float32", r.Float32(), r.Float32(), r.Float32())
show("Float64", r.Float64(), r.Float64(), r.Float64())
// ExpFloat64 values have an average of 1 but decay exponentially.
show("ExpFloat64", r.ExpFloat64(), r.ExpFloat64(), r.ExpFloat64())
// NormFloat64 values have an average of 0 and a standard deviatior
show("NormFloat64", r.NormFloat64(), r.NormFloat64(), r.NormFloat64
// Int31, Int63, and Uint32 generate values of the given width.
// The Int method (not shown) is like either Int31 or Int63
// depending on the size of 'int'.
show("Int31", r.Int31(), r.Int31(), r.Int31())
show("Int63", r.Int63(), r.Int63(), r.Int63())
show("Uint32", r.Int63(), r.Int63(), r.Int63())
// Intn, Int31n, and Int63n limit their output to be < n.
// They do so more carefully than using r.Int()%n.
show("Intn(10)", r.Intn(10), r.Intn(10), r.Intn(10))
show("Int31n(10)", r.Int31n(10), r.Int31n(10), r.Int31n(10))
show("Int63n(10)", r.Int63n(10), r.Int63n(10), r.Int63n(10))
// Perm generates a random permutation of the numbers [0, n).
show("Perm", r.Perm(5), r.Perm(5), r.Perm(5))
```

Output:

```
Float32     0.2635776           0.6358173           0.6718283
Float64     0.628605430454327   0.4504798828572669  0.956275594937
ExpFloat64  0.3362240648200941  1.4256072328483647  0.243547588161
NormFloat64 0.17233959114940064 1.577014951434847   0.042591296411
Int31       1501292890          1486668269          182840835
Int63       3546343826724305832 5724354148158589552 52398467997066
Uint32      5927547564735367388 637072299495207830  41283119559582
Intn(10)    1                   2                   5
Int31n(10)  4                   7                   8
Int63n(10)  7                   6                   3
Perm        [1 4 2 3 0]         [4 2 1 3 0]         [1 2 4 0 3]
```

Index

Examples

type Source

```
type Source interface {
    Int63() int64
    Seed(seed int64)
}
```

Source代表一个生成均匀分布在范围[0, 1<<63)的int64值的（伪随机的）资源。

func NewSource

```
func NewSource(seed int64) Source
```

使用给定的种子创建一个伪随机资源。

type Rand

```
type Rand struct {
    // 内含隐藏或非导出字段
}
```

Rand生成服从多种分布的随机数。

func New

```
func New(src Source) *Rand
```

返回一个使用src生产的随机数来生成其他各种分布的随机数值的*Rand。

func (*Rand) Seed

```
func (r *Rand) Seed(seed int64)
```

使用给定的seed来初始化生成器到一个确定的状态。

func (*Rand) Int

```
func (r *Rand) Int() int
```

返回一个非负的伪随机int值。

func (*Rand) Int31

```
func (r *Rand) Int31() int32
```

返回一个int32类型的非负的31位伪随机数。

func (*Rand) Int63

```
func (r *Rand) Int63() int64
```

返回一个int64类型的非负的63位伪随机数。

func (*Rand) Uint32

```
func (r *Rand) Uint32() uint32
```

返回一个uint32类型的非负的32位伪随机数。

func (*Rand) Intn

```
func (r *Rand) Intn(n int) int
```

返回一个取值范围在[0,n)的伪随机int值，如果n<=0会panic。

func (*Rand) Int31n

```
func (r *Rand) Int31n(n int32) int32
```

返回一个取值范围在[0,n)的伪随机int32值，如果n<=0会panic

func (*Rand) Int63n

```
func (r *Rand) Int63n(n int64) int64
```

返回一个取值范围在[0,n)的伪随机int64值，如果n<=0会panic。

func (*Rand) Float32

```
func (r *Rand) Float32() float32
```

返回一个取值范围在[0.0, 1.0)的伪随机float32值。

func (*Rand) Float64

```
func (r *Rand) Float64() float64
```

返回一个取值范围在[0.0, 1.0)的伪随机float64值。

func (*Rand) NormFloat64

```
func (r *Rand) NormFloat64() float64
```

返回一个服从标准正态分布（标准差=1，期望=0）、取值范围在[-math.MaxFloat64, +math.MaxFloat64]的float64值

如果要生成不同的正态分布值，调用者可用如下代码调整输出：

```
sample = NormFloat64() * 标准差 + 期望
```

func (*Rand) ExpFloat64

```
func (r *Rand) ExpFloat64() float64
```

返回一个服从标准指数分布（率参数=1，率参数是期望的倒数）、取值范围在(0, +math.MaxFloat64]的float64值

如要生成不同的指数分布值，调用者可用如下代码调整输出：

```
sample = ExpFloat64() / 率参数
```

func (*Rand) Perm

```
func (r *Rand) Perm(n int) []int
```

返回一个有n个元素的，[0,n)范围内整数的伪随机排列的切片。

type Zipf

```
type Zipf struct {
    // 内含隐藏或非导出字段
}
```

Zipf生成服从齐普夫分布的随机数。

func NewZipf

```
func NewZipf(r *Rand, s float64, v float64, imax uint64) *Zipf
```

NewZipf返回一个[0, imax]范围内的齐普夫随机数生成器。

齐普夫分布：值k出现的几率p(k)正比于(v+k)**(-s)，其中s>1且k>=0且v>=1。

func (*Zipf) Uint64

```
func (z *Zipf) Uint64() uint64
```

Uint64返回一个服从Zipf对象描述的齐普夫分布的随机数。

func Seed

```
func Seed(seed int64)
```

使用给定的seed将默认资源初始化到一个确定的状态；如未调用Seed，默认资源的行为就好像调用了Seed(1)。

func Int

```
func Int() int
```

返回一个非负的伪随机int值。

func Int31

```
func Int31() int32
```

返回一个int32类型的非负的31位伪随机数。

func Int63

```
func Int63() int64
```

返回一个int64类型的非负的63位伪随机数。

func Uint32

```
func Uint32() uint32
```

返回一个uint32类型的非负的32位伪随机数。

func Intn

```
func Intn(n int) int
```

返回一个取值范围在[0,n)的伪随机int值，如果n<=0会panic。

func Int31n

```
func Int31n(n int32) int32
```

返回一个取值范围在[0,n)的伪随机int32值，如果n<=0会panic。

func Int63n

```
func Int63n(n int64) int64
```

返回一个取值范围在[0, n)的伪随机int64值，如果n<=0会panic。

func Float32

```
func Float32() float32
```

返回一个取值范围在[0.0, 1.0)的伪随机float32值。

func Float64

```
func Float64() float64
```

返回一个取值范围在[0.0, 1.0)的伪随机float64值。

func NormFloat64

```
func NormFloat64() float64
```

返回一个服从标准正态分布（标准差=1，期望=0）、取值范围在[-math.MaxFloat64, +math.MaxFloat64]的float64值

如果要生成不同的正态分布值，调用者可用如下代码调整输出：

```
sample = NormFloat64() * 标准差 + 期望
```

func ExpFloat64

```
func ExpFloat64() float64
```

返回一个服从标准指数分布（率参数=1，率参数是期望的倒数）、取值范围在(0, +math.MaxFloat64]的float64值

如要生成不同的指数分布值，调用者可用如下代码调整输出：

```
sample = ExpFloat64() / 率参数
```

func Perm

```
func Perm(n int) []int
```

返回一个有n个元素的，[0,n)范围内整数的伪随机排列的切片。

package mime

```
import "mime"
```

mime实现了MIME的部分规定。

Index

func AddExtensionType

```
func AddExtensionType(ext, typ string) error
```

函数将扩展名和mimetype建立偶联；扩展名应以点号开始，例如".html"。

func FormatMediaType

```
func FormatMediaType(t string, param map[string]string) string
```

函数根据RFC 2045和 RFC 2616的规定将媒体类型t和参数param连接为一个mime媒体类型，类型和参数都采用小写字母。任一个参数不合法都会返回空字符串。

func ParseMediaType

```
func ParseMediaType(v string) (mediatype string, params map[string]
```

函数根据RFC 1521解析一个媒体类型值以及可能的参数。媒体类型值一般应为Content-Type和Conten-Disposition头域的值（参见RFC 2183）。成功的调用会返回小写字母、去空格的媒体类型和一个非空的map。返回的map映射小写字母的属性和对应的属性值。

func TypeByExtension

```
func TypeByExtension(ext string) string
```

函数返回与扩展名偶联的MIME类型。扩展名应以点号开始，如".html"。如果扩展名未偶联类型，函数会返回""。

内建的偶联表很小，但在unix系统会从本地系统的一或多个mime.types文件（参加下表）进行增补。

```
/etc/mime.types
/etc/apache2/mime.types
/etc/apache/mime.types
```

Windows系统的mime类型从注册表获取。文本类型的字符集参数默认设置为"utf-8"。

package multipart

```
import "mime/multipart"
```

multipart实现了MIME的multipart解析，参见RFC 2046。该实现适用于HTTP（RFC 2388）和常见浏览器生成的multipart主体。

Index

Examples

type File

```
type File interface {
    io.Reader
    io.ReaderAt
    io.Seeker
    io.Closer
}
```

File是一个接口，实现了对一个multipart信息中文件记录的访问。它的内容可以保持在内存或者硬盘中，如果保持在硬盘中，底层类型就会是*os.File。

type FileHeader

```
type FileHeader struct {
    Filename string
    Header   textproto.MIMEHeader
    // 内含隐藏或非导出字段
}
```

FileHeader描述一个multipart请求的（一个）文件记录的信息。

func (*FileHeader) Open

```
func (fh *FileHeader) Open() (File, error)
```

Open方法打开并返回其关联的文件。

type Part

```
type Part struct {
    // 主体的头域，如果存在，是按Go的http.Header风格标准化的，如"foo-bar"己
    // 有一个特殊情况，如果"Content-Transfer-Encoding"头的值是"quoted-p
    // 该头将从本map中隐藏，而主体会在调用Read时透明的解码。
    Header textproto.MIMEHeader
    // 内含隐藏或非导出字段
}
```

Part代表multipart主体的单独一个记录。

func (*Part) FileName

```
func (p *Part) FileName() string
```

返回Part 的Content-Disposition 头的文件名参数。

func (*Part) FormName

```
func (p *Part) FormName() string
```

如果p的Content-Disposition头值为"form-data"，则返回名字参数；否则返回空字符串。

func (*Part) Read

```
func (p *Part) Read(d []byte) (n int, err error)
```

Read方法读取一个记录的主体，也就是其头域之后到下一记录之前的部分。

func (*Part) Close

```
func (p *Part) Close() error
```

type Form

```
type Form struct {
    Value map[string][]string
    File  map[string][]*FileHeader
}
```

Form是一个解析过的multipart表格。它的File参数部分保存在内存或者硬盘上，可以使用*FileHeader类型属性值的Open方法访问。它的Value 参数部分保存为字符串，两者都以属性名为键。

func (*Form) RemoveAll

```
func (f *Form) RemoveAll() error
```

删除Form关联的所有临时文件。

type Reader

```
type Reader struct {
    // 内含隐藏或非导出字段
}
```

Reader是MIME的multipart主体所有记录的迭代器。Reader的底层会根据需要解析输入，不支持Seek。

func NewReader

```
func NewReader(r io.Reader, boundary string) *Reader
```

函数使用给出的MIME边界和r创建一个multipart读取器。

边界一般从信息的"Content-Type" 头的"boundary"属性获取。可使用mime.ParseMediaType函数解析这种头域。

Example

```
msg := &mail.Message{
    Header: map[string][]string{
        "Content-Type": []string{"multipart/mixed; boundary=foo"},
    },
    Body: strings.NewReader(
        "--foo\r\nFoo: one\r\n\r\nA section\r\n" +
            "--foo\r\nFoo: two\r\n\r\nAnd another\r\n" +
            "--foo--\r\n"),
}
mediaType, params, err := mime.ParseMediaType(msg.Header.Get("Conte
if err != nil {
    log.Fatal(err)
}
if strings.HasPrefix(mediaType, "multipart/") {
    mr := multipart.NewReader(msg.Body, params["boundary"])
    for {
        p, err := mr.NextPart()
        if err == io.EOF {
            return
        }
        if err != nil {
            log.Fatal(err)
        }
        slurp, err := ioutil.ReadAll(p)
        if err != nil {
            log.Fatal(err)
        }
        fmt.Printf("Part %q: %q\n", p.Header.Get("Foo"), slurp)
    }
}
```

Output:

```
Part "one": "A section"
Part "two": "And another"
```

func (*Reader) ReadForm

```
func (r *Reader) ReadForm(maxMemory int64) (f *Form, err error)
```

ReadForm解析整个multipart信息中所有Content-Disposition头的值为"form-data"的记录。它会把最多maxMemory字节的文件记录保存在内存里，其余保存在硬盘的临时文件里。

func (*Reader) NextPart

```
func (r *Reader) NextPart() (*Part, error)
```

NextPart返回multipart的下一个记录或者返回错误。如果没有更多记录会返回io.EOF。

type Writer

```
type Writer struct {
    // 内含隐藏或非导出字段
}
```

Writer类型用于生成multipart信息。

func NewWriter

```
func NewWriter(w io.Writer) *Writer
```

NewWriter函数返回一个设定了一个随机边界的Writer，数据写入w。

func (*Writer) FormDataContentType

```
func (w *Writer) FormDataContentType() string
```

方法返回w对应的HTTP multipart请求的Content-Type的值，多以multipart/form-data起始。

func (*Writer) Boundary

```
func (w *Writer) Boundary() string
```

方法返回该Writer的边界。

func (*Writer) SetBoundary

```
func (w *Writer) SetBoundary(boundary string) error
```

SetBoundary方法重写Writer默认的随机生成的边界为提供的boundary参数。方法必须在创建任何记录之前调用，boundary只能包含特定的ascii字符，并且长度应在1-69字节之间。

func (*Writer) CreatePart

```
func (w *Writer) CreatePart(header textproto.MIMEHeader) (io.Writer
```

CreatePart方法使用提供的header创建一个新的multipart记录。该记录的主体应该写入返回的Writer接口。调用本方法后，任何之前的记录都不能再写入。

func (*Writer) CreateFormField

```
func (w *Writer) CreateFormField(fieldname string) (io.Writer, erro
```

CreateFormField方法使用给出的属性名调用CreatePart方法。

func (*Writer) CreateFormFile

```
func (w *Writer) CreateFormFile(fieldname, filename string) (io.Wri
```

CreateFormFile是CreatePart方法的包装， 使用给出的属性名和文件名创建一个新的form-data头。

func (*Writer) WriteField

```
func (w *Writer) WriteField(fieldname, value string) error
```

WriteField方法调用CreateFormField并写入给出的value。

func (*Writer) Close

```
func (w *Writer) Close() error
```

Close方法结束multipart信息，并将结尾的边界写入底层io.Writer接口。

package net

```
import "net"
```

net包提供了可移植的网络I/O接口，包括TCP/IP、UDP、域名解析和Unix域socket。

虽然本包提供了对网络原语的访问，大部分使用者只需要Dial、Listen和Accept函数提供的基本接口；以及相关的Conn和Listener接口。crypto/tls包提供了相同的接口和类似的Dial和Listen函数。

Dial函数和服务端建立连接：

```
conn, err := net.Dial("tcp", "google.com:80")
if err != nil {
    // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := bufio.NewReader(conn).ReadString('\n')
// ...
```

Listen函数创建的服务端：

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
    // handle error
}
for {
    conn, err := ln.Accept()
    if err != nil {
        // handle error
        continue
    }
    go handleConnection(conn)
}
```

Index

Examples


```
const (
    IPv4len = 4
    IPv6len = 16
)
```

IP address lengths (bytes).

Variables

```
var (
    IPv4bcast     = IPv4(255, 255, 255, 255) // 广播地址
    IPv4allsys    = IPv4(224, 0, 0, 1)       // 所有主机和路由器
    IPv4allrouter = IPv4(224, 0, 0, 2)       // 所有路由器
    IPv4zero      = IPv4(0, 0, 0, 0)         // 本地地址，只能作为源地
)
```

常用的IPv4地址。

```
var (
    IPv6zero                   = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (
    IPv6unspecified            = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (
    IPv6loopback               = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (
    IPv6interfacelocalallnodes = IP{0xff, 0x01, 0, 0, 0, 0, 0, 0, (
    IPv6linklocalallnodes      = IP{0xff, 0x02, 0, 0, 0, 0, 0, 0, (
    IPv6linklocalallrouters    = IP{0xff, 0x02, 0, 0, 0, 0, 0, 0, (
)
```

常用的IPv6地址。

```
var (
    ErrWriteToConnected = errors.New("use of WriteTo with pre-conne
)
```

很多OpError类型的错误会包含本错误。

type ParseError

```
type ParseError struct {
    Type string
    Text string
}
```

ParseError代表一个格式错误的字符串，Type为期望的格式。

func (*ParseError) Error

```
func (e *ParseError) Error() string
```

type Error

```
type Error interface {
    error
    Timeout() bool   // 错误是否为超时？
    Temporary() bool // 错误是否是临时的？
}
```

Error代表一个网络错误。

type UnknownNetworkError

```
type UnknownNetworkError string
```

func (UnknownNetworkError) Error

```
func (e UnknownNetworkError) Error() string
```

func (UnknownNetworkError) Temporary

```
func (e UnknownNetworkError) Temporary() bool
```

func (UnknownNetworkError) Timeout

```
func (e UnknownNetworkError) Timeout() bool
```

type InvalidAddrError

```
type InvalidAddrError string
```

func (InvalidAddrError) Error

```
func (e InvalidAddrError) Error() string
```

func (InvalidAddrError) Temporary

```
func (e InvalidAddrError) Temporary() bool
```

func (InvalidAddrError) Timeout

```
func (e InvalidAddrError) Timeout() bool
```

type DNSConfigError

```
type DNSConfigError struct {
    Err error
}
```

DNSConfigError代表读取主机DNS配置时出现的错误。

func (*DNSConfigError) Error

```
func (e *DNSConfigError) Error() string
```

func (*DNSConfigError) Temporary

```
func (e *DNSConfigError) Temporary() bool
```

func (*DNSConfigError) Timeout

```
func (e *DNSConfigError) Timeout() bool
```

type DNSError

```
type DNSError struct {
    Err       string // 错误的描述
    Name      string // 查询的名称
    Server    string // 使用的服务器
    IsTimeout bool
}
```

DNSError代表DNS查询的错误。

func (*DNSError) Error

```
func (e *DNSError) Error() string
```

func (*DNSError) Temporary

```
func (e *DNSError) Temporary() bool
```

func (*DNSError) Timeout

```
func (e *DNSError) Timeout() bool
```

type AddrError

```
type AddrError struct {
    Err  string
    Addr string
}
```

func (*AddrError) Error

```
func (e *AddrError) Error() string
```

func (*AddrError) Temporary

```
func (e *AddrError) Temporary() bool
```

func (*AddrError) Timeout

```
func (e *AddrError) Timeout() bool
```

type OpError

```
type OpError struct {
    // Op是出现错误的操作，如"read"或"write"
    Op  string
    // Net是错误所在的网络类型，如"tcp"或"udp6"
    Net string
    // Addr是出现错误的网络地址
    Addr Addr
    // Err是操作中出现的错误
    Err error
}
```

OpError是经常被net包的函数返回的错误类型。它描述了该错误的操作、网络类型和网络地址。

func (*OpError) Error

```
func (e *OpError) Error() string
```

func (*OpError) Temporary

```
func (e *OpError) Temporary() bool
```

func (*OpError) Timeout

```
func (e *OpError) Timeout() bool
```

func SplitHostPort

```
func SplitHostPort(hostport string) (host, port string, err error)
```

函数将格式为"host:port"、"[host]:port"或"[ipv6-host%zone]:port"的网络地址分割为host或ipv6-host%zone和port两个部分。Ipv6的文字地址或者主机名必须用方括号括起来，如"[::1]:80"、"[ipv6-host]:http"、"[ipv6-host%zone]:80"。

func JoinHostPort

```
func JoinHostPort(host, port string) string
```

函数将host和port合并为一个网络地址。一般格式为"host:port"；如果host含有冒号或百分号，格式为"[host]:port"。

type HardwareAddr

```
type HardwareAddr []byte
```

HardwareAddr类型代表一个硬件地址（MAC地址）。

func ParseMAC

```
func ParseMAC(s string) (hw HardwareAddr, err error)
```

ParseMAC函数使用如下格式解析一个IEEE 802 MAC-48、EUI-48或EUI-64硬件地址：

```
01:23:45:67:89:ab
01:23:45:67:89:ab:cd:ef
01-23-45-67-89-ab
01-23-45-67-89-ab-cd-ef
0123.4567.89ab
0123.4567.89ab.cdef
```

func (HardwareAddr) String

```
func (a HardwareAddr) String() string
```

type Flags

```
type Flags uint
```

```
const (
    FlagUp           Flags = 1 << iota // 接口在活动状态
    FlagBroadcast                      // 接口支持广播
    FlagLoopback                       // 接口是环回的
    FlagPointToPoint                   // 接口是点对点的
    FlagMulticast                      // 接口支持组播
)
```

func (Flags) String

```
func (f Flags) String() string
```

type Interface

```
type Interface struct {
    Index        int          // 索引，>=1的整数
    MTU          int          // 最大传输单元
    Name         string       // 接口名，例如"en0"、"lo0"、"eth0.100"
    HardwareAddr HardwareAddr // 硬件地址，IEEE MAC-48、EUI-48或EUI-6
    Flags        Flags        // 接口的属性，例如FlagUp、FlagLoopback、
}
```

Interface类型代表一个网络接口（系统与网络的一个接点）。包含接口索引到名字的映射，也包含接口的设备信息。

func InterfaceByIndex

```
func InterfaceByIndex(index int) (*Interface, error)
```

InterfaceByIndex返回指定索引的网络接口。

func InterfaceByName

```
func InterfaceByName(name string) (*Interface, error)
```

InterfaceByName返回指定名字的网络接口。

func (*Interface) Addrs

```
func (ifi *Interface) Addrs() ([]Addr, error)
```

Addrs方法返回网络接口ifi的一或多个接口地址。

func (*Interface) MulticastAddrs

```
func (ifi *Interface) MulticastAddrs() ([]Addr, error)
```

MulticastAddrs返回网络接口ifi加入的多播组地址。

func Interfaces

```
func Interfaces() ([]Interface, error)
```

Interfaces返回该系统的网络接口列表。

func InterfaceAddrs

```
func InterfaceAddrs() ([]Addr, error)
```

InterfaceAddrs返回该系统的网络接口的地址列表。

type IP

```
type IP []byte
```

IP类型是代表单个IP地址的[]byte切片。本包的函数都可以接受4字节（IPv4）和16字节（IPv6）的切片作为输入。

注意，IP地址是IPv4地址还是IPv6地址是语义上的属性，而不取决于切片的长度：16字节的切片也可以是IPv4地址。

func IPv4

```
func IPv4(a, b, c, d byte) IP
```

IPv4返回包含一个IPv4地址a.b.c.d的IP地址（16字节格式）。

func ParseIP

```
func ParseIP(s string) IP
```

ParseIP将s解析为IP地址，并返回该地址。如果s不是合法的IP地址文本表示，ParseIP会返回nil。

字符串可以是小数点分隔的IPv4格式（如"74.125.19.99"）或IPv6格式（如"2001:4860:0:2001::68"）格式。

func (IP) IsGlobalUnicast

```
func (ip IP) IsGlobalUnicast() bool
```

如果ip是全局单播地址，则返回真。

func (IP) IsLinkLocalUnicast

```
func (ip IP) IsLinkLocalUnicast() bool
```

如果ip是链路本地单播地址，则返回真。

func (IP) IsInterfaceLocalMulticast

```
func (ip IP) IsInterfaceLocalMulticast() bool
```

如果ip是接口本地组播地址，则返回真。

func (IP) IsLinkLocalMulticast

```
func (ip IP) IsLinkLocalMulticast() bool
```

如果ip是链路本地组播地址，则返回真。

func (IP) IsMulticast

```
func (ip IP) IsMulticast() bool
```

如果ip是组播地址，则返回真。

func (IP) IsLoopback

```
func (ip IP) IsLoopback() bool
```

如果ip是环回地址，则返回真。

func (IP) IsUnspecified

```
func (ip IP) IsUnspecified() bool
```

如果ip是未指定地址，则返回真。

func (IP) DefaultMask

```
func (ip IP) DefaultMask() IPMask
```

函数返回IP地址ip的默认子网掩码。只有IPv4有默认子网掩码；如果ip不是合法的IPv4地址，会返回nil。

func (IP) Equal

```
func (ip IP) Equal(x IP) bool
```

如果ip和x代表同一个IP地址，Equal会返回真。代表同一地址的IPv4地址和IPv6地址也被认为是相等的。

func (IP) To16

```
func (ip IP) To16() IP
```

To16将一个IP地址转换为16字节表示。如果ip不是一个IP地址（长度错误），To16会返回nil。

func (IP) To4

```
func (ip IP) To4() IP
```

To4将一个IPv4地址转换为4字节表示。如果ip不是IPv4地址，To4会返回nil。

func (IP) Mask

```
func (ip IP) Mask(mask IPMask) IP
```

Mask方法认为mask为ip的子网掩码，返回ip的网络地址部分的ip。（主机地址部分都置0）

func (IP) String

```
func (ip IP) String() string
```

String返回IP地址ip的字符串表示。如果ip是IPv4地址，返回值的格式为点分隔的，如"74.125.19.99"；否则表示为IPv6格式，如"2001:4860:0:2001::68"。

func (IP) MarshalText

```
func (ip IP) MarshalText() ([]byte, error)
```

MarshalText实现了encoding.TextMarshaler接口，返回值和String方法一样。

func (*IP) UnmarshalText

```
func (ip *IP) UnmarshalText(text []byte) error
```

UnmarshalText实现了encoding.TextUnmarshaler接口。IP地址字符串应该是ParseIP函数可以接受的格式。

type IPMask

```
type IPMask []byte
```

IPMask代表一个IP地址的掩码。

func IPv4Mask

```
func IPv4Mask(a, b, c, d byte) IPMask
```

IPv4Mask返回一个4字节格式的IPv4掩码a.b.c.d。

func CIDRMask

```
func CIDRMask(ones, bits int) IPMask
```

CIDRMask返回一个IPMask类型值，该返回值总共有bits个字位，其中前ones个字位都是1，其余字位都是0。

func (IPMask) Size

```
func (m IPMask) Size() (ones, bits int)
```

Size返回m的前导的1字位数和总字位数。如果m不是规范的子网掩码（字位：/^1+0+$/），将返会(0, 0)。

func (IPMask) String

```
func (m IPMask) String() string
```

String返回m的十六进制格式，没有标点。

type IPNet

```
type IPNet struct {
    IP   IP     // 网络地址
    Mask IPMask // 子网掩码
}
```

IPNet表示一个IP网络。

func ParseCIDR

```
func ParseCIDR(s string) (IP, *IPNet, error)
```

ParseCIDR将s作为一个CIDR（无类型域间路由）的IP地址和掩码字符串，如"192.168.100.1/24"或"2001:DB8::/48"，解析并返回IP地址和IP网络，参见RFC 4632和RFC 4291。

本函数会返回IP地址和该IP所在的网络和掩码。例如，ParseCIDR("192.168.100.1/16")会返回IP地址192.168.100.1和IP网络192.168.0.0/16。

func (*IPNet) Contains

```
func (n *IPNet) Contains(ip IP) bool
```

Contains报告该网络是否包含地址ip。

func (*IPNet) Network

```
func (n *IPNet) Network() string
```

Network返回网络类型名："ip+net"，注意该类型名是不合法的。

func (*IPNet) String

```
func (n *IPNet) String() string
```

String返回n的CIDR表示，如"192.168.100.1/24"或"2001:DB8::/48"，参见RFC 4632和RFC 4291。如果n的Mask字段不是规范格式，它会返回一个包含n.IP.String()、斜线、n.Mask.String()（此时表示为无标点十六进制格式）的字符串，如"192.168.100.1/c000ff00"。

type Addr

```
type Addr interface {
    Network() string // 网络名
    String() string  // 字符串格式的地址
}
```

Addr代表一个网络终端地址。

type Conn

```
type Conn interface {
    // Read从连接中读取数据
    // Read方法可能会在超过某个固定时间限制后超时返回错误，该错误的Timeout()
    Read(b []byte) (n int, err error)
    // Write从连接中写入数据
    // Write方法可能会在超过某个固定时间限制后超时返回错误，该错误的Timeout(
    Write(b []byte) (n int, err error)
    // Close方法关闭该连接
    // 并会导致任何阻塞中的Read或Write方法不再阻塞并返回错误
    Close() error
    // 返回本地网络地址
    LocalAddr() Addr
    // 返回远端网络地址
    RemoteAddr() Addr
    // 设定该连接的读写deadline，等价于同时调用SetReadDeadline和SetWrite
    // deadline是一个绝对时间，超过该时间后I/O操作就会直接因超时失败返回而不
    // deadline对之后的所有I/O操作都起效，而不仅仅是下一次的读或写操作
    // 参数t为零值表示不设置期限
    SetDeadline(t time.Time) error
    // 设定该连接的读操作deadline，参数t为零值表示不设置期限
    SetReadDeadline(t time.Time) error
    // 设定该连接的写操作deadline，参数t为零值表示不设置期限
    // 即使写入超时，返回值n也可能>0，说明成功写入了部分数据
    SetWriteDeadline(t time.Time) error
}
```

Conn接口代表通用的面向流的网络连接。多个线程可能会同时调用同一个Conn的方法。

func Dial

```
func Dial(network, address string) (Conn, error)
```

在网络network上连接地址address，并返回一个Conn接口。可用的网络类型有：

"tcp"、"tcp4"、"tcp6"、"udp"、"udp4"、"udp6"、"ip"、"ip4"、"ip6"、"unix"、"unixgram"、"unixpacket"

对TCP和UDP网络，地址格式是host:port或[host]:port，参见函数JoinHostPort和SplitHostPort。

```
Dial("tcp", "12.34.56.78:80")
Dial("tcp", "google.com:http")
Dial("tcp", "[2001:db8::1]:http")
Dial("tcp", "[fe80::1%lo0]:80")
```

对IP网络，network必须是"ip"、"ip4"、"ip6"后跟冒号和协议号或者协议名，地址必须是IP地址字面值。

```
Dial("ip4:1", "127.0.0.1")
Dial("ip6:ospf", "::1")
```

对Unix网络，地址必须是文件系统路径。

func DialTimeout

```
func DialTimeout(network, address string, timeout time.Duration) (C
```

DialTimeout类似Dial但采用了超时。timeout参数如果必要可包含名称解析。

func Pipe

```
func Pipe() (Conn, Conn)
```

Pipe创建一个内存中的同步、全双工网络连接。连接的两端都实现了Conn接口。一端的读取对应另一端的写入，直接将数据在两端之间作拷贝；没有内部缓冲。

type PacketConn

```
type PacketConn interface {
    // ReadFrom方法从连接读取一个数据包，并将有效信息写入b
    // ReadFrom方法可能会在超过某个固定时间限制后超时返回错误，该错误的Timeo
    // 返回写入的字节数和该数据包的来源地址
    ReadFrom(b []byte) (n int, addr Addr, err error)
    // WriteTo方法将有效数据b写入一个数据包发送给addr
    // WriteTo方法可能会在超过某个固定时间限制后超时返回错误，该错误的Timeou
    // 在面向数据包的连接中，写入超时非常罕见
    WriteTo(b []byte, addr Addr) (n int, err error)
    // Close方法关闭该连接
    // 会导致任何阻塞中的ReadFrom或WriteTo方法不再阻塞并返回错误
    Close() error
    // 返回本地网络地址
    LocalAddr() Addr
    // 设定该连接的读写deadline
    SetDeadline(t time.Time) error
    // 设定该连接的读操作deadline，参数t为零值表示不设置期限
    // 如果时间到达deadline，读操作就会直接因超时失败返回而不会阻塞
    SetReadDeadline(t time.Time) error
    // 设定该连接的写操作deadline，参数t为零值表示不设置期限
    // 如果时间到达deadline，写操作就会直接因超时失败返回而不会阻塞
    // 即使写入超时，返回值n也可能>0，说明成功写入了部分数据
    SetWriteDeadline(t time.Time) error
}
```

PacketConn接口代表通用的面向数据包的网络连接。多个线程可能会同时调用同一个Conn的方法。

func ListenPacket

```
func ListenPacket(net, laddr string) (PacketConn, error)
```

ListenPacket函数监听本地网络地址laddr。网络类型net必须是面向数据包的网络类型：

"ip"、"ip4"、"ip6"、"udp"、"udp4"、"udp6"、或"unixgram"。laddr的格式参见Dial函数。

type Dialer

```
type Dialer struct {
    // Timeout是dial操作等待连接建立的最大时长，默认值代表没有超时。
    // 如果Deadline字段也被设置了，dial操作也可能更早失败。
    // 不管有没有设置超时，操作系统都可能强制执行它的超时设置。
    // 例如，TCP（系统）超时一般在3分钟左右。
    Timeout time.Duration
    // Deadline是一个具体的时间点期限，超过该期限后，dial操作就会失败。
    // 如果Timeout字段也被设置了，dial操作也可能更早失败。
    // 零值表示没有期限，即遵守操作系统的超时设置。
    Deadline time.Time
    // LocalAddr是dial一个地址时使用的本地地址。
    // 该地址必须是与dial的网络相容的类型。
    // 如果为nil，将会自动选择一个本地地址。
    LocalAddr Addr
    // DualStack允许单次dial操作在网络类型为"tcp"，
    // 且目的地是一个主机名的DNS记录具有多个地址时，
    // 尝试建立多个IPv4和IPv6连接，并返回第一个建立的连接。
    DualStack bool
    // KeepAlive指定一个活动的网络连接的生命周期；如果为0，会禁止keep-alive
    // 不支持keep-alive的网络连接会忽略本字段。
    KeepAlive time.Duration
}
```

Dialer类型包含与某个地址建立连接时的参数。

每一个字段的零值都等价于没有该字段。因此调用Dialer零值的Dial方法等价于调用Dial函数。

func (*Dialer) Dial

```
func (d *Dialer) Dial(network, address string) (Conn, error)
```

Dial在指定的网络上连接指定的地址。参见Dial函数获取网络和地址参数的描述。

type Listener

```
type Listener interface {
    // Addr返回该接口的网络地址
    Addr() Addr
    // Accept等待并返回下一个连接到该接口的连接
    Accept() (c Conn, err error)
    // Close关闭该接口，并使任何阻塞的Accept操作都会不再阻塞并返回错误。
    Close() error
}
```

Listener是一个用于面向流的网络协议的公用的网络监听器接口。多个线程可能会同时调用一个Listener的方法。

Example

```
// Listen on TCP port 2000 on all interfaces.
l, err := net.Listen("tcp", ":2000")
if err != nil {
    log.Fatal(err)
}
defer l.Close()
for {
    // Wait for a connection.
    conn, err := l.Accept()
    if err != nil {
        log.Fatal(err)
    }
    // Handle the connection in a new goroutine.
    // The loop then returns to accepting, so that
    // multiple connections may be served concurrently.
    go func(c net.Conn) {
        // Echo all incoming data.
        io.Copy(c, c)
        // Shut down the connection.
        c.Close()
    }(conn)
}
```

func Listen

```
func Listen(net, laddr string) (Listener, error)
```

返回在一个本地网络地址laddr上监听的Listener。网络类型参数net必须是面向流的网络：

"tcp"、"tcp4"、"tcp6"、"unix"或"unixpacket"。参见Dial函数获取laddr的语法。

type IPAddr

```
type IPAddr struct {
    IP   IP
    Zone string // IPv6范围寻址域
}
```

IPAddr代表一个IP终端的地址。

func ResolveIPAddr

```
func ResolveIPAddr(net, addr string) (*IPAddr, error)
```

ResolveIPAddr将addr作为一个格式为"host"或"ipv6-host%zone"的IP地址来解析。函数会在参数net指定的网络类型上解析，net必须是"ip"、"ip4"或"ip6"。

func (*IPAddr) Network

```
func (a *IPAddr) Network() string
```

Network返回地址的网络类型："ip"。

func (*IPAddr) String

```
func (a *IPAddr) String() string
```

type TCPAddr

```
type TCPAddr struct {
    IP   IP
    Port int
    Zone string // IPv6范围寻址域
}
```

TCPAddr代表一个TCP终端地址。

func ResolveTCPAddr

```
func ResolveTCPAddr(net, addr string) (*TCPAddr, error)
```

ResolveTCPAddr将addr作为TCP地址解析并返回。参数addr格式为"host:port"或"[ipv6-host%zone]:port"，解析得到网络名和端口名；net必须是"tcp"、"tcp4"或"tcp6"。

IPv6地址字面值/名称必须用方括号包起来，如"[::1]:80"、"[ipv6-host]:http"或"[ipv6-host%zone]:80"。

func (*TCPAddr) Network

```
func (a *TCPAddr) Network() string
```

返回地址的网络类型，"tcp"。

func (*TCPAddr) String

```
func (a *TCPAddr) String() string
```

type UDPAddr

```
type UDPAddr struct {
    IP   IP
    Port int
    Zone string // IPv6范围寻址域
}
```

UDPAddr代表一个UDP终端地址。

func ResolveUDPAddr

```
func ResolveUDPAddr(net, addr string) (*UDPAddr, error)
```

ResolveTCPAddr将addr作为TCP地址解析并返回。参数addr格式为"host:port"或"[ipv6-host%zone]:port"，解析得到网络名和端口名；net必须是"udp"、"udp4"或"udp6"。

IPv6地址字面值/名称必须用方括号包起来，如"[::1]:80"、"[ipv6-host]:http"或"[ipv6-host%zone]:80"。

func (*UDPAddr) Network

```
func (a *UDPAddr) Network() string
```

返回地址的网络类型，"udp"。

func (*UDPAddr) String

```
func (a *UDPAddr) String() string
```

type UnixAddr

```
type UnixAddr struct {
    Name string
    Net  string
}
```

UnixAddr代表一个Unix域socket终端地址。

func ResolveUnixAddr

```
func ResolveUnixAddr(net, addr string) (*UnixAddr, error)
```

ResolveUnixAddr将addr作为Unix域socket地址解析，参数net指定网络类型："unix"、"unixgram"或"unixpacket"。

func (*UnixAddr) Network

```
func (a *UnixAddr) Network() string
```

返回地址的网络类型，"unix"，"unixgram"或"unixpacket"。

func (*UnixAddr) String

```
func (a *UnixAddr) String() string
```

type IPConn

```
type IPConn struct {
    // 内含隐藏或非导出字段
}
```

IPConn类型代表IP网络连接，实现了Conn和PacketConn接口。

func DialIP

```
func DialIP(netProto string, laddr, raddr *IPAddr) (*IPConn, error)
```

DialIP在网络协议netProto上连接本地地址laddr和远端地址raddr，netProto必须是"ip"、"ip4"或"ip6"后跟冒号和协议名或协议号。

func ListenIP

```
func ListenIP(netProto string, laddr *IPAddr) (*IPConn, error)
```

ListenIP创建一个接收目的地是本地地址laddr的IP数据包的网络连接，返回的*IPConn的ReadFrom和WriteTo方法可以用来发送和接收IP数据包。（每个包都可获取来源址或者设置目标地址）

func (*IPConn) LocalAddr

```
func (c *IPConn) LocalAddr() Addr
```

LocalAddr返回本地网络地址

func (*IPConn) RemoteAddr

```
func (c *IPConn) RemoteAddr() Addr
```

RemoteAddr返回远端网络地址

func (*IPConn) SetReadBuffer

```
func (c *IPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer设置该连接的系统接收缓冲

func (*IPConn) SetWriteBuffer

```
func (c *IPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer设置该连接的系统发送缓冲

func (*IPConn) SetDeadline

```
func (c *IPConn) SetDeadline(t time.Time) error
```

SetDeadline设置读写操作绝对期限，实现了Conn接口的SetDeadline方法

func (*IPConn) SetReadDeadline

```
func (c *IPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline设置读操作绝对期限，实现了Conn接口的SetReadDeadline方法

func (*IPConn) SetWriteDeadline

```
func (c *IPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline设置写操作绝对期限，实现了Conn接口的SetWriteDeadline方法

func (*IPConn) Read

```
func (c *IPConn) Read(b []byte) (int, error)
```

Read实现Conn接口Read方法

func (*IPConn) ReadFrom

```
func (c *IPConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom实现PacketConn接口ReadFrom方法。注意本方法有bug，应避免使用。

func (*IPConn) ReadFromIP

```
func (c *IPConn) ReadFromIP(b []byte) (int, *IPAddr, error)
```

ReadFromIP从c读取一个IP数据包，将有效负载拷贝到b，返回拷贝字节数和数据包来源地址。

ReadFromIP方法会在超过一个固定的时间点之后超时，并返回一个错误。注意本方法有bug，应避免使用。

func (*IPConn) ReadMsgIP

```
func (c *IPConn) ReadMsgIP(b, oob []byte) (n, oobn, flags int, addr
```

ReadMsgIP从c读取一个数据包，将有效负载拷贝进b，相关的带外数据拷贝进oob，返回拷贝进b的字节数，拷贝进oob的字节数，数据包的flag，数据包来源地址和可能的错误。

func (*IPConn) Write

```
func (c *IPConn) Write(b []byte) (int, error)
```

Write实现Conn接口Write方法

func (*IPConn) WriteTo

```
func (c *IPConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo实现PacketConn接口WriteTo方法

func (*IPConn) WriteToIP

```
func (c *IPConn) WriteToIP(b []byte, addr *IPAddr) (int, error)
```

WriteToIP通过c向地址addr发送一个数据包，b为包的有效负载，返回写入的字节。

WriteToIP方法会在超过一个固定的时间点之后超时，并返回一个错误。在面向数据包的连接上，写入超时是十分罕见的。

func (*IPConn) WriteMsgIP

```
func (c *IPConn) WriteMsgIP(b, oob []byte, addr *IPAddr) (n, oobn i
```

WriteMsgIP通过c向地址addr发送一个数据包，b和oob分别为包有效负载和对应的带外数据，返回写入的字节数（包数据、带外数据）和可能的错误。

func (*IPConn) Close

```
func (c *IPConn) Close() error
```

Close关闭连接

func (*IPConn) File

```
func (c *IPConn) File() (f *os.File, err error)
```

File方法设置下层的os.File为阻塞模式并返回其副本。

使用者有责任在用完后关闭f。关闭c不影响f，关闭f也不影响c。返回的os.File类型文件描述符和原本的网络连接是不同的。试图使用该副本修改本体的属性可能会（也可能不会）得到期望的效果。

type TCPConn

```
type TCPConn struct {
    // 内含隐藏或非导出字段
}
```

TCPConn代表一个TCP网络连接，实现了Conn接口。

func DialTCP

```
func DialTCP(net string, laddr, raddr *TCPAddr) (*TCPConn, error)
```

DialTCP在网络协议net上连接本地地址laddr和远端地址raddr。net必须是"tcp"、"tcp4"、"tcp6"；如果laddr不是nil，将使用它作为本地地址，否则自动选择一个本地地址。

func (*TCPConn) LocalAddr

```
func (c *TCPConn) LocalAddr() Addr
```

LocalAddr返回本地网络地址

func (*TCPConn) RemoteAddr

```
func (c *TCPConn) RemoteAddr() Addr
```

RemoteAddr返回远端网络地址

func (*TCPConn) SetReadBuffer

```
func (c *TCPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer设置该连接的系统接收缓冲

func (*TCPConn) SetWriteBuffer

```
func (c *TCPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer设置该连接的系统发送缓冲

func (*TCPConn) SetDeadline

```
func (c *TCPConn) SetDeadline(t time.Time) error
```

SetDeadline设置读写操作期限，实现了Conn接口的SetDeadline方法

func (*TCPConn) SetReadDeadline

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline设置读操作期限，实现了Conn接口的SetReadDeadline方法

func (*TCPConn) SetWriteDeadline

```
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline设置写操作期限，实现了Conn接口的SetWriteDeadline方法

func (*TCPConn) SetKeepAlive

```
func (c *TCPConn) SetKeepAlive(keepalive bool) error
```

SetKeepAlive设置操作系统是否应该在该连接中发送keepalive信息

func (*TCPConn) SetKeepAlivePeriod

```
func (c *TCPConn) SetKeepAlivePeriod(d time.Duration) error
```

SetKeepAlivePeriod设置keepalive的周期，超出会断开

func (*TCPConn) SetLinger

```
func (c *TCPConn) SetLinger(sec int) error
```

SetLinger设定当连接中仍有数据等待发送或接受时的Close方法的行为。

如果sec < 0（默认），Close方法立即返回，操作系统停止后台数据发送；如果sec == 0，Close立刻返回，操作系统丢弃任何未发送或未接收的数据；如果sec > 0，Close方法阻塞最多sec秒，等待数据发送或者接收，在一些操作系统中，在超时后，任何未发送的数据会被丢弃。

func (*TCPConn) SetNoDelay

```
func (c *TCPConn) SetNoDelay(noDelay bool) error
```

SetNoDelay设定操作系统是否应该延迟数据包传递，以便发送更少的数据包（Nagle's算法）。默认为真，即数据应该在Write方法后立刻发送。

func (*TCPConn) Read

```
func (c *TCPConn) Read(b []byte) (int, error)
```

Read实现了Conn接口Read方法

func (*TCPConn) Write

```
func (c *TCPConn) Write(b []byte) (int, error)
```

Write实现了Conn接口Write方法

func (*TCPConn) ReadFrom

```
func (c *TCPConn) ReadFrom(r io.Reader) (int64, error)
```

ReadFrom实现了io.ReaderFrom接口的ReadFrom方法

func (*TCPConn) Close

```
func (c *TCPConn) Close() error
```

Close关闭连接

func (*TCPConn) CloseRead

```
func (c *TCPConn) CloseRead() error
```

CloseRead关闭TCP连接的读取侧（以后不能读取），应尽量使用Close方法。

func (*TCPConn) CloseWrite

```
func (c *TCPConn) CloseWrite() error
```

CloseWrite关闭TCP连接的写入侧（以后不能写入），应尽量使用Close方法。

func (*TCPConn) File

```
func (c *TCPConn) File() (f *os.File, err error)
```

File方法设置下层的os.File为阻塞模式并返回其副本。

使用者有责任在用完后关闭f。关闭c不影响f，关闭f也不影响c。返回的os.File类型文件描述符和原本的网络连接是不同的。试图使用该副本修改本体的属性可能会（也可能不会）得到期望的效果。

type UDPConn

```
type UDPConn struct {
    // 内含隐藏或非导出字段
}
```

UDPConn代表一个UDP网络连接，实现了Conn和PacketConn接口。

func DialUDP

```
func DialUDP(net string, laddr, raddr *UDPAddr) (*UDPConn, error)
```

DialTCP在网络协议net上连接本地地址laddr和远端地址raddr。net必须是"udp"、"udp4"、"udp6"；如果laddr不是nil，将使用它作为本地地址，否则自动选择一个本地地址。

func ListenUDP

```
func ListenUDP(net string, laddr *UDPAddr) (*UDPConn, error)
```

ListenUDP创建一个接收目的地是本地地址laddr的UDP数据包的网络连接。net必须是"udp"、"udp4"、"udp6"；如果laddr端口为0，函数将选择一个当前可用的端口，可以用Listener的Addr方法获得该端口。返回的*UDPConn的ReadFrom和WriteTo方法可以用来发送和接收UDP数据包（每个包都可获得来源地址或设置目标地址）。

func ListenMulticastUDP

```
func ListenMulticastUDP(net string, ifi *Interface, gaddr *UDPAddr)
```

ListenMulticastUDP接收目的地是ifi接口上的组地址gaddr的UDP数据包。它指定了使用的接口，如果ifi是nil，将使用默认接口。

func (*UDPConn) LocalAddr

```
func (c *UDPConn) LocalAddr() Addr
```

LocalAddr返回本地网络地址

func (*UDPConn) RemoteAddr

```
func (c *UDPConn) RemoteAddr() Addr
```

RemoteAddr返回远端网络地址

func (*UDPConn) SetReadBuffer

```
func (c *UDPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer设置该连接的系统接收缓冲

func (*UDPConn) SetWriteBuffer

```
func (c *UDPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer设置该连接的系统发送缓冲

func (*UDPConn) SetDeadline

```
func (c *UDPConn) SetDeadline(t time.Time) error
```

SetDeadline设置读写操作期限，实现了Conn接口的SetDeadline方法

func (*UDPConn) SetReadDeadline

```
func (c *UDPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline设置读操作期限，实现了Conn接口的SetReadDeadline方法

func (*UDPConn) SetWriteDeadline

```
func (c *UDPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline设置写操作期限，实现了Conn接口的SetWriteDeadline方法

func (*UDPConn) Read

```
func (c *UDPConn) Read(b []byte) (int, error)
```

Read实现Conn接口Read方法

func (*UDPConn) ReadFrom

```
func (c *UDPConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom实现PacketConn接口ReadFrom方法

func (*UDPConn) ReadFromUDP

```
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err
```

ReadFromUDP从c读取一个UDP数据包，将有效负载拷贝到b，返回拷贝字节数和数据包来源地址。

ReadFromUDP方法会在超过一个固定的时间点之后超时，并返回一个错误。

func (*UDPConn) ReadMsgUDP

```
func (c *UDPConn) ReadMsgUDP(b, oob []byte) (n, oobn, flags int, ad
```

ReadMsgUDP从c读取一个数据包，将有效负载拷贝进b，相关的带外数据拷贝进oob，返回拷贝进b的字节数，拷贝进oob的字节数，数据包的flag，数据包来源地址和可能的错误。

func (*UDPConn) Write

```
func (c *UDPConn) Write(b []byte) (int, error)
```

Write实现Conn接口Write方法

func (*UDPConn) WriteTo

```
func (c *UDPConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo实现PacketConn接口WriteTo方法

func (*UDPConn) WriteToUDP

```
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int, error)
```

WriteToUDP通过c向地址addr发送一个数据包，b为包的有效负载，返回写入的字节。

WriteToUDP方法会在超过一个固定的时间点之后超时，并返回一个错误。在面向数据包的连接上，写入超时是十分罕见的。

func (*UDPConn) WriteMsgUDP

```
func (c *UDPConn) WriteMsgUDP(b, oob []byte, addr *UDPAddr) (n, oob
```

WriteMsgUDP通过c向地址addr发送一个数据包，b和oob分别为包有效负载和对应的带外数据，返回写入的字节数（包数据、带外数据）和可能的错误。

func (*UDPConn) Close

```
func (c *UDPConn) Close() error
```

Close关闭连接

func (*UDPConn) File

```
func (c *UDPConn) File() (f *os.File, err error)
```

File方法设置下层的os.File为阻塞模式并返回其副本。

使用者有责任在用完后关闭f。关闭c不影响f，关闭f也不影响c。返回的os.File类型文件描述符和原本的网络连接是不同的。试图使用该副本修改本体的属性可能会（也可能不会）得到期望的效果。

type UnixConn

```
type UnixConn struct {
    // 内含隐藏或非导出字段
}
```

UnixConn代表Unix域socket连接，实现了Conn和PacketConn接口。

func DialUnix

```
func DialUnix(net string, laddr, raddr *UnixAddr) (*UnixConn, erroı
```

DialUnix在网络协议net上连接本地地址laddr和远端地址raddr。net必须是"unix"、"unixgram"、"unixpacket"，如果laddr不是nil将使用它作为本地地址，否则自动选择一个本地地址。

func ListenUnixgram

```
func ListenUnixgram(net string, laddr *UnixAddr) (*UnixConn, error)
```

ListenUnixgram接收目的地是本地地址laddr的Unix datagram网络连接。net必须是"unixgram"，返回的*UnixConn的ReadFrom和WriteTo方法可以用来发送和接收数据包（每个包都可获取来源址或者设置目标地址）。

func (*UnixConn) LocalAddr

```
func (c *UnixConn) LocalAddr() Addr
```

LocalAddr返回本地网络地址

func (*UnixConn) RemoteAddr

```
func (c *UnixConn) RemoteAddr() Addr
```

RemoteAddr返回远端网络地址

func (*UnixConn) SetReadBuffer

```
func (c *UnixConn) SetReadBuffer(bytes int) error
```

SetReadBuffer设置该连接的系统接收缓冲

func (*UnixConn) SetWriteBuffer

```
func (c *UnixConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer设置该连接的系统发送缓冲

func (*UnixConn) SetDeadline

```
func (c *UnixConn) SetDeadline(t time.Time) error
```

SetDeadline设置读写操作期限，实现了Conn接口的SetDeadline方法

func (*UnixConn) SetReadDeadline

```
func (c *UnixConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline设置读操作期限，实现了Conn接口的SetReadDeadline方法

func (*UnixConn) SetWriteDeadline

```
func (c *UnixConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline设置写操作期限，实现了Conn接口的SetWriteDeadline方法

func (*UnixConn) Read

```
func (c *UnixConn) Read(b []byte) (int, error)
```

Read实现了Conn接口Read方法

func (*UnixConn) ReadFrom

```
func (c *UnixConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom实现PacketConn接口ReadFrom方法

func (*UnixConn) ReadFromUnix

```
func (c *UnixConn) ReadFromUnix(b []byte) (n int, addr *UnixAddr, e
```

ReadFromUnix从c读取一个UDP数据包，将有效负载拷贝到b，返回拷贝字节数和数据包来源地址。

ReadFromUnix方法会在超过一个固定的时间点之后超时，并返回一个错误。

func (*UnixConn) ReadMsgUnix

```
func (c *UnixConn) ReadMsgUnix(b, oob []byte) (n, oobn, flags int,
```

ReadMsgUnix从c读取一个数据包，将有效负载拷贝进b，相关的带外数据拷贝进oob，返回拷贝进b的字节数，拷贝进oob的字节数，数据包的flag，数据包来源地址和可能的错误。

func (*UnixConn) Write

```
func (c *UnixConn) Write(b []byte) (int, error)
```

Write实现了Conn接口Write方法

func (*UnixConn) WriteTo

```
func (c *UnixConn) WriteTo(b []byte, addr Addr) (n int, err error)
```

WriteTo实现PacketConn接口WriteTo方法

func (*UnixConn) WriteToUnix

```
func (c *UnixConn) WriteToUnix(b []byte, addr *UnixAddr) (n int, er
```

WriteToUnix通过c向地址addr发送一个数据包，b为包的有效负载，返回写入的字节。

WriteToUnix方法会在超过一个固定的时间点之后超时，并返回一个错误。在面向数据包的连接上，写入超时是十分罕见的。

func (*UnixConn) WriteMsgUnix

```
func (c *UnixConn) WriteMsgUnix(b, oob []byte, addr *UnixAddr) (n,
```

WriteMsgUnix通过c向地址addr发送一个数据包，b和oob分别为包有效负载和对应的带外数据，返回写入的字节数（包数据、带外数据）和可能的错误。

func (*UnixConn) Close

```
func (c *UnixConn) Close() error
```

Close关闭连接

func (*UnixConn) CloseRead

```
func (c *UnixConn) CloseRead() error
```

CloseRead关闭TCP连接的读取侧（以后不能读取），应尽量使用Close方法

func (*UnixConn) CloseWrite

```
func (c *UnixConn) CloseWrite() error
```

CloseWrite关闭TCP连接的写入侧（以后不能写入），应尽量使用Close方法

func (*UnixConn) File

```
func (c *UnixConn) File() (f *os.File, err error)
```

File方法设置下层的os.File为阻塞模式并返回其副本。

使用者有责任在用完后关闭f。关闭c不影响f，关闭f也不影响c。返回的os.File类型文件描述符和原本的网络连接是不同的。试图使用该副本修改本体的属性可能会（也可能不会）得到期望的效果。

type TCPListener

```
type TCPListener struct {
    // 内含隐藏或非导出字段
}
```

TCPListener代表一个TCP网络的监听者。使用者应尽量使用Listener接口而不是假设（网络连接为）TCP。

func ListenTCP

```
func ListenTCP(net string, laddr *TCPAddr) (*TCPListener, error)
```

ListenTCP在本地TCP地址laddr上声明并返回一个*TCPListener，net参数必须是"tcp"、"tcp4"、"tcp6"，如果laddr的端口字段为0，函数将选择一个当前可用的端口，可以用Listener的Addr方法获得该端口。

func (*TCPListener) Addr

```
func (l *TCPListener) Addr() Addr
```

Addr返回l监听的的网络地址，一个*TCPAddr。

func (*TCPListener) SetDeadline

```
func (l *TCPListener) SetDeadline(t time.Time) error
```

设置监听器执行的期限，t为Time零值则会关闭期限限制。

func (*TCPListener) Accept

```
func (l *TCPListener) Accept() (Conn, error)
```

Accept用于实现Listener接口的Accept方法；他会等待下一个呼叫，并返回一个该呼叫的Conn接口。

func (*TCPListener) AcceptTCP

```
func (l *TCPListener) AcceptTCP() (*TCPConn, error)
```

AcceptTCP接收下一个呼叫，并返回一个新的*TCPConn。

func (*TCPListener) Close

```
func (l *TCPListener) Close() error
```

Close停止监听TCP地址，已经接收的连接不受影响。

func (*TCPListener) File

```
func (l *TCPListener) File() (f *os.File, err error)
```

File方法返回下层的os.File的副本，并将该副本设置为阻塞模式。

使用者有责任在用完后关闭f。关闭c不影响f，关闭f也不影响c。返回的os.File类型文件描述符和原本的网络连接是不同的。试图使用该副本修改本体的属性可能会（也可能不会）得到期望的效果。

type UnixListener

```
type UnixListener struct {
    // 内含隐藏或非导出字段
}
```

UnixListener代表一个Unix域scoket的监听者。使用者应尽量使用Listener接口而不是假设（网络连接为）Unix域scoket。

func ListenUnix

```
func ListenUnix(net string, laddr *UnixAddr) (*UnixListener, error)
```

ListenTCP在Unix域scoket地址laddr上声明并返回一个*UnixListener，net参数必须是"unix"或"unixpacket"。

func (*UnixListener) Addr

```
func (l *UnixListener) Addr() Addr
```

Addr返回l的监听的Unix域socket地址

func (*UnixListener) SetDeadline

```
func (l *UnixListener) SetDeadline(t time.Time) (err error)
```

设置监听器执行的期限，t为Time零值则会关闭期限限制

func (*UnixListener) Accept

```
func (l *UnixListener) Accept() (c Conn, err error)
```

Accept用于实现Listener接口的Accept方法；他会等待下一个呼叫，并返回一个该呼叫的Conn接口。

func (*UnixListener) AcceptUnix

```
func (l *UnixListener) AcceptUnix() (*UnixConn, error)
```

AcceptUnix接收下一个呼叫，并返回一个新的*UnixConn。

func (*UnixListener) Close

```
func (l *UnixListener) Close() error
```

Close停止监听Unix域socket地址，已经接收的连接不受影响。

func (*UnixListener) File

```
func (l *UnixListener) File() (f *os.File, err error)
```

File方法返回下层的os.File的副本，并将该副本设置为阻塞模式。

使用者有责任在用完后关闭f。关闭c不影响f，关闭f也不影响c。返回的os.File类型文件描述符和原本的网络连接是不同的。试图使用该副本修改本体的属性可能会（也可能不会）得到期望的效果。

func FileConn

```
func FileConn(f *os.File) (c Conn, err error)
```

FileConn返回一个下层为文件f的网络连接的拷贝。调用者有责任在结束程序前关闭f。关闭c不会影响f，关闭f也不会影响c。本函数与各种实现了Conn接口的类型的File方法是对应的。

func FilePacketConn

```
func FilePacketConn(f *os.File) (c PacketConn, err error)
```

FilePacketConn函数返回一个下层为文件f的数据包网络连接的拷贝。调用者有责任在结束程序前关闭f。关闭c不会影响f，关闭f也不会影响c。本函数与各种实现了PacketConn接口的类型的File方法是对应的。

func FileListener

```
func FileListener(f *os.File) (l Listener, err error)
```

FileListener返回一个下层为文件f的网络监听器的拷贝。调用者有责任在使用结束后改变l。关闭l不会影响f，关闭f也不会影响l。本函数与各种实现了Listener接口的类型的File方法是对应的。

type MX

```
type MX struct {
    Host string
    Pref uint16
}
```

MX代表一条DNS MX记录（邮件交换记录），根据收信人的地址后缀来定位邮件服务器。

type NS

```
type NS struct {
    Host string
}
```

NS代表一条DNS NS记录（域名服务器记录），指定该域名由哪个DNS服务器来进行解析。

type SRV

```
type SRV struct {
    Target   string
    Port     uint16
    Priority uint16
    Weight   uint16
}
```

SRV代表一条DNS SRV记录（资源记录），记录某个服务由哪台计算机提供。

func LookupPort

```
func LookupPort(network, service string) (port int, err error)
```

LookupPort函数查询指定网络和服务的（默认）端口。

func LookupCNAME

```
func LookupCNAME(name string) (cname string, err error)
```

LookupCNAME函数查询name的规范DNS名（但该域名未必可以访问）。如果调用者不关心规范名可以直接调用LookupHost或者LookupIP；这两个函数都会在查询时考虑到规范名。

func LookupHost

```
func LookupHost(host string) (addrs []string, err error)
```

LookupHost函数查询主机的网络地址序列。

func LookupIP

```
func LookupIP(host string) (addrs []IP, err error)
```

LookupIP函数查询主机的ipv4和ipv6地址序列。

func LookupAddr

```
func LookupAddr(addr string) (name []string, err error)
```

LookupAddr查询某个地址，返回映射到该地址的主机名序列，本函数和LookupHost不互为反函数。

func LookupMX

```
func LookupMX(name string) (mx []*MX, err error)
```

LookupMX函数返回指定主机的按Pref字段排好序的DNS MX记录。

func LookupNS

```
func LookupNS(name string) (ns []*NS, err error)
```

LookupNS函数返回指定主机的DNS NS记录。

func LookupSRV

```
func LookupSRV(service, proto, name string) (cname string, addrs []
```

LookupSRV函数尝试执行指定服务、协议、主机的SRV查询。协议proto为"tcp"或"udp"。返回的记录按Priority字段排序，同一优先度按Weight字段随机排序。

LookupSRV函数按照RFC 2782的规定构建用于查询的DNS名。也就是说，它会查询_service._proto.name。为了适应将服务的SRV记录发布在非规范名下的情况，如果service和proto参数都是空字符串，函数会直接查询name。

func LookupTXT

```
func LookupTXT(name string) (txt []string, err error)
```

LookupTXT函数返回指定主机的DNS TXT记录。

Bugs

☞在任何POSIX平台上，从"ip4"网络使用ReadFrom或ReadFromIP方法读取数据时，即使有足够的空间，都可能不会返回完整的IPv4数据包，包括数据包的头域。即使Read或ReadMsgIP方法可以返回完整的数据包，也有可能出现这种情况。因为对go 1的兼容性要求，这个情况无法被修正。因此，当必须获取完整数据包时，建议你不要使用这两个方法，请使用Read或ReadMsgIP代替。

☞在OpenBSD系统中，在"tcp"网络监听时不会同时监听IPv4和IPv6连接。 因为该系统中IPv4通信不会导入IPv6套接字中。请使用两个独立的监听，如果有必要的话。

package http

`import "net/http"`

http包提供了HTTP客户端和服务端的实现。

Get、Head、Post和PostForm函数发出HTTP/ HTTPS请求。

```
resp, err := http.Get("http://example.com/")
...
resp, err := http.Post("http://example.com/upload", "image/jpeg", &
...
resp, err := http.PostForm("http://example.com/form",
    url.Values{"key": {"Value"}, "id": {"123"}})
```

程序在使用完回复后必须关闭回复的主体。

```
resp, err := http.Get("http://example.com/")
if err != nil {
    // handle error
}
defer resp.Body.Close()
body, err := ioutil.ReadAll(resp.Body)
// ...
```

要管理HTTP客户端的头域、重定向策略和其他设置，创建一个Client：

```
client := &http.Client{
    CheckRedirect: redirectPolicyFunc,
}
resp, err := client.Get("http://example.com")
// ...
req, err := http.NewRequest("GET", "http://example.com", nil)
// ...
req.Header.Add("If-None-Match", `W/"wyzzy"`)
resp, err := client.Do(req)
// ...
```

要管理代理、TLS配置、keep-alive、压缩和其他设置，创建一个Transport：

```
tr := &http.Transport{
    TLSClientConfig:    &tls.Config{RootCAs: pool},
    DisableCompression: true,
}
client := &http.Client{Transport: tr}
resp, err := client.Get("https://example.com")
```

Client和Transport类型都可以安全的被多个go程同时使用。出于效率考虑，应该一次建立、尽量重用。

ListenAndServe使用指定的监听地址和处理器启动一个HTTP服务端。处理器参数通常是nil，这表示采用包变量DefaultServeMux作为处理器。Handle和HandleFunc函数可以向DefaultServeMux添加处理器。

```
http.Handle("/foo", fooHandler)
http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})
log.Fatal(http.ListenAndServe(":8080", nil))
```

要管理服务端的行为，可以创建一个自定义的Server：

```
s := &http.Server{
    Addr:           ":8080",
    Handler:        myHandler,
    ReadTimeout:    10 * time.Second,
    WriteTimeout:   10 * time.Second,
    MaxHeaderBytes: 1 << 20,
}
log.Fatal(s.ListenAndServe())
```

Index

Examples

```
const (
    StatusContinue           = 100
    StatusSwitchingProtocols = 101
    StatusOK                   = 200
    StatusCreated              = 201
    StatusAccepted             = 202
    StatusNonAuthoritativeInfo = 203
    StatusNoContent            = 204
    StatusResetContent         = 205
    StatusPartialContent       = 206
    StatusMultipleChoices   = 300
    StatusMovedPermanently  = 301
    StatusFound             = 302
    StatusSeeOther          = 303
    StatusNotModified       = 304
    StatusUseProxy          = 305
    StatusTemporaryRedirect = 307
    StatusBadRequest                   = 400
    StatusUnauthorized                 = 401
    StatusPaymentRequired              = 402
    StatusForbidden                    = 403
    StatusNotFound                     = 404
    StatusMethodNotAllowed             = 405
    StatusNotAcceptable                = 406
    StatusProxyAuthRequired            = 407
    StatusRequestTimeout               = 408
    StatusConflict                     = 409
    StatusGone                         = 410
    StatusLengthRequired               = 411
    StatusPreconditionFailed           = 412
    StatusRequestEntityTooLarge        = 413
    StatusRequestURITooLong            = 414
    StatusUnsupportedMediaType         = 415
    StatusRequestedRangeNotSatisfiable = 416
    StatusExpectationFailed            = 417
    StatusTeapot                       = 418
    StatusInternalServerError     = 500
    StatusNotImplemented          = 501
    StatusBadGateway              = 502
    StatusServiceUnavailable      = 503
    StatusGatewayTimeout          = 504
    StatusHTTPVersionNotSupported = 505
)
```

HTTP状态码，参见RFC 2616

```
const DefaultMaxHeaderBytes = 1 << 20 // 1 MB
```

DefaultMaxHeaderBytes是HTTP请求的头域最大允许长度。可以通过设置Server.MaxHeaderBytes字段来覆盖。

```
const DefaultMaxIdleConnsPerHost = 2
```

DefaultMaxIdleConnsPerHost是Transport的MaxIdleConnsPerHost的默认值。

```
const TimeFormat = "Mon, 02 Jan 2006 15:04:05 GMT"
```

TimeFormat是当解析或生产HTTP头域中的时间时，用与time.Parse或time.Format函数的时间格式。这种格式类似time.RFC1123但强制采用GMT时区。

Variables

```
var (
    ErrHeaderTooLong        = &ProtocolError{"header too long"}
    ErrShortBody            = &ProtocolError{"entity body too short
    ErrNotSupported         = &ProtocolError{"feature not supportec
    ErrUnexpectedTrailer    = &ProtocolError{"trailer header withou
    ErrMissingContentLength = &ProtocolError{"missing ContentLength
    ErrNotMultipart         = &ProtocolError{"request Content-Type
    ErrMissingBoundary      = &ProtocolError{"no multipart boundary
)
```

HTTP请求的解析错误。

```
var (
    ErrWriteAfterFlush = errors.New("Conn.Write called after Flush'
    ErrBodyNotAllowed  = errors.New("http: request method or respor
    ErrHijacked        = errors.New("Conn has been hijacked")
    ErrContentLength   = errors.New("Conn.Write wrote more than the
)
```

会被HTTP服务端返回的错误。

```
var DefaultClient = &Client{}
```

DefaultClient是用于包函数Get、Head和Post的默认Client。

```
var DefaultServeMux = NewServeMux()
```

DefaultServeMux是用于Serve的默认ServeMux。

```
var ErrBodyReadAfterClose = errors.New("http: invalid Read on close
```

在Resquest或Response的Body字段已经关闭后，试图从中读取时，就会返回ErrBodyReadAfterClose。这个错误一般发生在：HTTP处理器中调用完ResponseWriter 接口的WriteHeader或Write后从请求中读取数据的时候。

```
var ErrHandlerTimeout = errors.New("http: Handler timeout")
```

在处理器超时以后调用ResponseWriter接口的Write方法，就会返回ErrHandlerTimeout。

```
var ErrLineTooLong = errors.New("header line too long")
```

```
var ErrMissingFile = errors.New("http: no such file")
```

当请求中没有提供给FormFile函数的文件字段名，或者该字段名不是文件字段时，该函数就会返回ErrMissingFile。

```
var ErrNoCookie = errors.New("http: named cookie not present")
```

```
var ErrNoLocation = errors.New("http: no Location header in respons
```

type ProtocolError

```
type ProtocolError struct {
    ErrorString string
}
```

HTTP请求解析错误。

func (*ProtocolError) Error

```
func (err *ProtocolError) Error() string
```

func CanonicalHeaderKey

```
func CanonicalHeaderKey(s string) string
```

CanonicalHeaderKey函数返回头域（表示为Header类型）的键s的规范化格式。规范化过程中让单词首字母和'-'后的第一个字母大写，其余字母小写。例如，"accept-encoding"规范化为"Accept-Encoding"。

func DetectContentType

```
func DetectContentType(data []byte) string
```

DetectContentType函数实现了http://mimesniff.spec.whatwg.org/描述的算法，用于确定数据的Content-Type。函数总是返回一个合法的MIME类型；如果它不能确定数据的类型，将返回"application/octet-stream"。它最多检查数据的前512字节。

func ParseHTTPVersion

```
func ParseHTTPVersion(vers string) (major, minor int, ok bool)
```

ParseHTTPVersion解析HTTP版本字符串。如"HTTP/1.0"返回(1, 0, true)。

func ParseTime

```
func ParseTime(text string) (t time.Time, err error)
```

ParseTime用3种格式TimeFormat, time.RFC850和time.ANSIC尝试解析一个时间头的值（如Date: header）。

func StatusText

```
func StatusText(code int) string
```

StatusText返回HTTP状态码code对应的文本，如220对应"OK"。如果code是未知的状态码，会返回""。

type ConnState

```
type ConnState int
```

ConnState代表一个客户端到服务端的连接的状态。本类型用于可选的Server.ConnState回调函数。

```
const (
    // StateNew代表一个新的连接，将要立刻发送请求。
    // 连接从这个状态开始，然后转变为StateAlive或StateClosed。
    StateNew ConnState = iota
    // StateActive代表一个已经读取了请求数据1到多个字节的连接。
    // 用于StateAlive的Server.ConnState回调函数在将连接交付给处理器之前被
    // 等到请求被处理完后，Server.ConnState回调函数再次被触发。
    // 在请求被处理后，连接状态改变为StateClosed、StateHijacked或StateId
    StateActive
    // StateIdle代表一个已经处理完了请求、处在闲置状态、等待新请求的连接。
    // 连接状态可以从StateIdle改变为StateActive或StateClosed。
    StateIdle
    // 代表一个被劫持的连接。这是一个终止状态，不会转变为StateClosed。
    StateHijacked
    // StateClosed代表一个关闭的连接。
    // 这是一个终止状态。被劫持的连接不会转变为StateClosed。
    StateClosed
)
```

func (ConnState) String

```
func (c ConnState) String() string
```

type Header

```
type Header map[string][]string
```

Header代表HTTP头域的键值对。

func (Header) Get

```
func (h Header) Get(key string) string
```

Get返回键对应的第一个值，如果键不存在会返回""。如要获取该键对应的值切片，请直接用规范格式的键访问map。

func (Header) Set

```
func (h Header) Set(key, value string)
```

Set添加键值对到h，如键已存在则会用只有新值一个元素的切片取代旧值切片。

func (Header) Add

```
func (h Header) Add(key, value string)
```

Add添加键值对到h，如键已存在则会将新的值附加到旧值切片后面。

func (Header) Del

```
func (h Header) Del(key string)
```

Del删除键值对。

func (Header) Write

```
func (h Header) Write(w io.Writer) error
```

Write以有线格式将头域写入w。

func (Header) WriteSubset

```
func (h Header) WriteSubset(w io.Writer, exclude map[string]bool) e
```

WriteSubset以有线格式将头域写入w。当exclude不为nil时，如果h的键值对的键在exclude中存在且其对应值为真，该键值对就不会被写入w。

type Cookie

```
type Cookie struct {
    Name       string
    Value      string
    Path       string
    Domain     string
    Expires    time.Time
    RawExpires string
    // MaxAge=0表示未设置Max-Age属性
    // MaxAge<0表示立刻删除该cookie，等价于"Max-Age: 0"
    // MaxAge>0表示存在Max-Age属性，单位是秒
    MaxAge   int
    Secure   bool
    HttpOnly bool
    Raw      string
    Unparsed []string // 未解析的“属性-值”对的原始文本
}
```

Cookie代表一个出现在HTTP回复的头域中Set-Cookie头的值里或者HTTP请求的头域中Cookie头的值里的HTTP cookie。

func (*Cookie) String

```
func (c *Cookie) String() string
```

String返回该cookie的序列化结果。如果只设置了Name和Value字段，序列化结果可用于HTTP请求的Cookie头或者HTTP回复的Set-Cookie头；如果设置了其他字段，序列化结果只能用于HTTP回复的Set-Cookie头。

type CookieJar

```
type CookieJar interface {
    // SetCookies管理从u的回复中收到的cookie
    // 根据其策略和实现，它可以选择是否存储cookie
    SetCookies(u *url.URL, cookies []*Cookie)
    // Cookies返回发送请求到u时应使用的cookie
    // 本方法有责任遵守RFC 6265规定的标准cookie限制
    Cookies(u *url.URL) []*Cookie
}
```

CookieJar管理cookie的存储和在HTTP请求中的使用。CookieJar的实现必须能安全的被多个go程同时使用。

net/http/cookiejar包提供了一个CookieJar的实现。

type Request

```
type Request struct {
    // Method指定HTTP方法（GET、POST、PUT等）。对客户端，""代表GET。
    Method string
    // URL在服务端表示被请求的URI，在客户端表示要访问的URL。
    //
    // 在服务端，URL字段是解析请求行的URI（保存在RequestURI字段）得到的，
    // 对大多数请求来说，除了Path和RawQuery之外的字段都是空字符串。
    // （参见RFC 2616, Section 5.1.2）
    //
    // 在客户端，URL的Host字段指定了要连接的服务器，
    // 而Request的Host字段（可选地）指定要发送的HTTP请求的Host头的值。
    URL *url.URL
    // 接收到的请求的协议版本。本包生产的Request总是使用HTTP/1.1
    Proto      string // "HTTP/1.0"
    ProtoMajor int    // 1
    ProtoMinor int    // 0
    // Header字段用来表示HTTP请求的头域。如果头域（多行键值对格式）为：
    //    accept-encoding: gzip, deflate
    //    Accept-Language: en-us
    //    Connection: keep-alive
    // 则：
    //    Header = map[string][]string{
    //        "Accept-Encoding": {"gzip, deflate"},
    //        "Accept-Language": {"en-us"},
    //        "Connection": {"keep-alive"},
    //    }
    // HTTP规定头域的键名（头名）是大小写敏感的，请求的解析器通过规范化头域的
    // 在客户端的请求，可能会被自动添加或重写Header中的特定的头，参见Request
    Header Header
    // Body是请求的主体。
    //
    // 在客户端，如果Body是nil表示该请求没有主体买入GET请求。
    // Client的Transport字段会负责调用Body的Close方法。
    //
    // 在服务端，Body字段总是非nil的；但在没有主体时，读取Body会立刻返回EOF。
    // Server会关闭请求的主体，ServeHTTP处理器不需要关闭Body字段。
    Body io.ReadCloser
    // ContentLength记录相关内容的长度。
    // 如果为-1，表示长度未知，如果>=0，表示可以从Body字段读取ContentLengt
    // 在客户端，如果Body非nil而该字段为0，表示不知道Body的长度。
    ContentLength int64
    // TransferEncoding按从最外到最里的顺序列出传输编码，空切片表示"identi
    // 本字段一般会被忽略。当发送或接受请求时，会自动添加或移除"chunked"传输
    TransferEncoding []string
    // Close在服务端指定是否在回复请求后关闭连接，在客户端指定是否在发送请求后
    Close bool
    // 在服务端，Host指定URL会在其上寻找资源的主机。
    // 根据RFC 2616，该值可以是Host头的值，或者URL自身提供的主机名。
    // Host的格式可以是"host:port"。
```

```
    //
    // 在客户端，请求的Host字段（可选地）用来重写请求的Host头。
    // 如过该字段为""，Request.Write方法会使用URL字段的Host。
    Host string
    // Form是解析好的表单数据，包括URL字段的query参数和POST或PUT的表单数据。
    // 本字段只有在调用ParseForm后才有效。在客户端，会忽略请求中的本字段而使用
    Form url.Values
    // PostForm是解析好的POST或PUT的表单数据。
    // 本字段只有在调用ParseForm后才有效。在客户端，会忽略请求中的本字段而使用
    PostForm url.Values
    // MultipartForm是解析好的多部件表单，包括上传的文件。
    // 本字段只有在调用ParseMultipartForm后才有效。
    // 在客户端，会忽略请求中的本字段而使用Body替代。
    MultipartForm *multipart.Form
    // Trailer指定了会在请求主体之后发送的额外的头域。
    //
    // 在服务端，Trailer字段必须初始化为只有trailer键，所有键都对应nil值。
    // （客户端会声明哪些trailer会发送）
    // 在处理器从Body读取时，不能使用本字段。
    // 在从Body的读取返回EOF后，Trailer字段会被更新完毕并包含非nil的值。
    // （如果客户端发送了这些键值对），此时才可以访问本字段。
    //
    // 在客户端，Trail必须初始化为一个包含将要发送的键值对的映射。（值可以是n
    // ContentLength字段必须是0或-1，以启用"chunked"传输编码发送请求。
    // 在开始发送请求后，Trailer可以在读取请求主体期间被修改，
    // 一旦请求主体返回EOF，调用者就不可再修改Trailer。
    //
    // 很少有HTTP客户端、服务端或代理支持HTTP trailer。
    Trailer Header
    // RemoteAddr允许HTTP服务器和其他软件记录该请求的来源地址，一般用于日志。
    // 本字段不是ReadRequest函数填写的，也没有定义格式。
    // 本包的HTTP服务器会在调用处理器之前设置RemoteAddr为"IP:port"格式的地
    // 客户端会忽略请求中的RemoteAddr字段。
    RemoteAddr string
    // RequestURI是被客户端发送到服务端的请求的请求行中未修改的请求URI
    // （参见RFC 2616, Section 5.1）
    // 一般应使用URI字段，在客户端设置请求的本字段会导致错误。
    RequestURI string
    // TLS字段允许HTTP服务器和其他软件记录接收到该请求的TLS连接的信息
    // 本字段不是ReadRequest函数填写的。
    // 对启用了TLS的连接，本包的HTTP服务器会在调用处理器之前设置TLS字段，否
    // 客户端会忽略请求中的TLS字段。
    TLS *tls.ConnectionState
}
```

Request类型代表一个服务端接受到的或者客户端发送出去的HTTP请求。

Request各字段的意义和用途在服务端和客户端是不同的。除了字段本身上方文档，还可参见Request.Write方法和RoundTripper接口的文档。

func NewRequest

```
func NewRequest(method, urlStr string, body io.Reader) (*Request, e
```

NewRequest使用指定的方法、网址和可选的主题创建并返回一个新的*Request。

如果body参数实现了io.Closer接口，Request返回值的Body 字段会被设置为body，并会被Client类型的Do、Post和PostFOrm方法以及Transport.RoundTrip方法关闭。

func ReadRequest

```
func ReadRequest(b *bufio.Reader) (req *Request, err error)
```

ReadRequest从b读取并解析出一个HTTP请求。（本函数主要用在服务端从下层获取请求）

func (*Request) ProtoAtLeast

```
func (r *Request) ProtoAtLeast(major, minor int) bool
```

ProtoAtLeast报告该请求使用的HTTP协议版本至少是major.minor。

func (*Request) UserAgent

```
func (r *Request) UserAgent() string
```

UserAgent返回请求中的客户端用户代理信息（请求的User-Agent头）。

func (*Request) Referer

```
func (r *Request) Referer() string
```

Referer返回请求中的访问来路信息。（请求的Referer头）

Referer在请求中就是拼错了的，这是HTTP早期就有的错误。该值也可以从用Header["Referer"]获取；让获取Referer字段变成方法的好处是，编译器可以诊断使用正确单词拼法的req.Referrer()的程序，但却不能诊断使用Header["Referrer"]的程序。

func (*Request) AddCookie

```
func (r *Request) AddCookie(c *Cookie)
```

AddCookie向请求中添加一个cookie。按照RFC 6265 section 5.4的跪地，AddCookie不会添加超过一个Cookie头字段。这表示所有的cookie都写在同一行，用分号分隔（cookie内部用逗号分隔属性）。

func (*Request) SetBasicAuth

```
func (r *Request) SetBasicAuth(username, password string)
```

SetBasicAuth使用提供的用户名和密码，采用HTTP基本认证，设置请求的Authorization头。HTTP基本认证会明码传送用户名和密码。

func (*Request) Write

```
func (r *Request) Write(w io.Writer) error
```

Write方法以有线格式将HTTP/1.1请求写入w（用于将请求写入下层TCPConn等）。本方法会考虑请求的如下字段：

```
Host
URL
Method (defaults to "GET")
Header
ContentLength
TransferEncoding
Body
```

如果存在Body，ContentLength字段<= 0且TransferEncoding字段未显式设置为["identity"]，Write方法会显式添加"Transfer-Encoding: chunked"到请求的头域。Body字段会在发送完请求后关闭。

func (*Request) WriteProxy

```
func (r *Request) WriteProxy(w io.Writer) error
```

WriteProxy类似Write但会将请求以HTTP代理期望的格式发送。

尤其是，按照RFC 2616 Section 5.1.2，WriteProxy会使用绝对URI（包括协议和主机名）来初始化请求的第1行（Request-URI行）。无论何种情况，WriteProxy都会使用r.Host或r.URL.Host设置Host头。

func (*Request) Cookies

```
func (r *Request) Cookies() []*Cookie
```

Cookies解析并返回该请求的Cookie头设置的cookie。

func (*Request) Cookie

```
func (r *Request) Cookie(name string) (*Cookie, error)
```

Cookie返回请求中名为name的cookie，如果未找到该cookie会返回nil, ErrNoCookie。

func (*Request) ParseForm

```
func (r *Request) ParseForm() error
```

ParseForm解析URL中的查询字符串，并将解析结果更新到r.Form字段。

对于POST或PUT请求，ParseForm还会将body当作表单解析，并将结果既更新到r.PostForm也更新到r.Form。解析结果中，POST或PUT请求主体要优先于URL查询字符串（同名变量，主体的值在查询字符串的值前面）。

如果请求的主体的大小没有被MaxBytesReader函数设定限制，其大小默认限制为开头10MB。

ParseMultipartForm会自动调用ParseForm。重复调用本方法是无意义的。

func (*Request) ParseMultipartForm

```
func (r *Request) ParseMultipartForm(maxMemory int64) error
```

ParseMultipartForm将请求的主体作为multipart/form-data解析。请求的整个主体都会被解析，得到的文件记录最多maxMemery字节保存在内存，其余部分保存在硬盘的temp文件里。如果必要，ParseMultipartForm会自行调用ParseForm。重复调用本方法是无意义的。

func (*Request) FormValue

```
func (r *Request) FormValue(key string) string
```

FormValue返回key为键查询r.Form字段得到结果[]string切片的第一个值。POST和PUT主体中的同名参数优先于URL查询字符串。如果必要，本函数会隐式调用ParseMultipartForm和ParseForm。

func (*Request) PostFormValue

```
func (r *Request) PostFormValue(key string) string
```

PostFormValue返回key为键查询r.PostForm字段得到结果[]string切片的第一个值。如果必要，本函数会隐式调用ParseMultipartForm和ParseForm。

func (*Request) FormFile

```
func (r *Request) FormFile(key string) (multipart.File, *multipart
```

FormFile返回以key为键查询r.MultipartForm字段得到结果中的第一个文件和它的信息。如果必要，本函数会隐式调用ParseMultipartForm和ParseForm。查询失败会返回ErrMissingFile错误。

func (*Request) MultipartReader

```
func (r *Request) MultipartReader() (*multipart.Reader, error)
```

如果请求是multipart/form-data POST请求，MultipartReader返回一个multipart.Reader接口，否则返回nil和一个错误。使用本函数代替ParseMultipartForm，可以将r.Body作为流处理。

type Response

```
type Response struct {
    Status     string // 例如"200 OK"
    StatusCode int    // 例如200
    Proto      string // 例如"HTTP/1.0"
    ProtoMajor int    // 例如1
    ProtoMinor int    // 例如0
    // Header保管头域的键值对。
    // 如果回复中有多个头的键相同，Header中保存为该键对应用逗号分隔串联起来的
    // （参见RFC 2616 Section 4.2）
    // 被本结构体中的其他字段复制保管的头（如ContentLength）会从Header中删
    //
    // Header中的键都是规范化的，参见CanonicalHeaderKey函数
    Header Header
    // Body代表回复的主体。
    // Client类型和Transport类型会保证Body字段总是非nil的，即使回复没有主体
    // 关闭主体是调用者的责任。
    // 如果服务端采用"chunked"传输编码发送的回复，Body字段会自动进行解码。
    Body io.ReadCloser
    // ContentLength记录相关内容的长度。
    // 其值为-1表示长度未知（采用chunked传输编码）
    // 除非对应的Request.Method是"HEAD"，其值>=0表示可以从Body读取的字节数
    ContentLength int64
    // TransferEncoding按从最外到最里的顺序列出传输编码，空切片表示"identi
    TransferEncoding []string
    // Close记录头域是否指定应在读取完主体后关闭连接。（即Connection头）
    // 该值是给客户端的建议，Response.Write方法的ReadResponse函数都不会关
    Close bool
    // Trailer字段保存和头域相同格式的trailer键值对，和Header字段相同类型
    Trailer Header
    // Request是用来获取此回复的请求
    // Request的Body字段是nil（因为已经被用掉了）
    // 这个字段是被Client类型发出请求并获得回复后填充的
    Request *Request
    // TLS包含接收到该回复的TLS连接的信息。 对未加密的回复，本字段为nil。
    // 返回的指针是被（同一TLS连接接收到的）回复共享的，不应被修改。
    TLS *tls.ConnectionState
}
```

Response代表一个HTTP请求的回复。

func ReadResponse

```
func ReadResponse(r *bufio.Reader, req *Request) (*Response, error)
```

ReadResponse从r读取并返回一个HTTP 回复。req参数是可选的，指定该回复对应的请求（即是对该请求的回复）。如果是nil，将假设请求是GET请求。客户端必须在结束resp.Body的读取后关闭它。读取完毕并关闭后，客户端可以检查

resp.Trailer字段获取回复的trailer的键值对。（本函数主要用在客户端从下层获取回复）

func (*Response) ProtoAtLeast

```
func (r *Response) ProtoAtLeast(major, minor int) bool
```

ProtoAtLeast报告该回复使用的HTTP协议版本至少是major.minor。

func (*Response) Cookies

```
func (r *Response) Cookies() []*Cookie
```

Cookies解析并返回该回复中的Set-Cookie头设置的cookie。

func (*Response) Location

```
func (r *Response) Location() (*url.URL, error)
```

Location返回该回复的Location头设置的URL。相对地址的重定向会相对于该回复对应的请求来确定绝对地址。如果回复中没有Location头，会返回nil, ErrNoLocation。

func (*Response) Write

```
func (r *Response) Write(w io.Writer) error
```

Write以有线格式将回复写入w（用于将回复写入下层TCPConn等）。本方法会考虑如下字段：

```
StatusCode
ProtoMajor
ProtoMinor
Request.Method
TransferEncoding
Trailer
Body
ContentLength
Header（不规范的键名和它对应的值会导致不可预知的行为）
```

Body字段在发送完回复后会被关闭。

type ResponseWriter

```
type ResponseWriter interface {
    // Header返回一个Header类型值，该值会被WriteHeader方法发送。
    // 在调用WriteHeader或Write方法后再改变该对象是没有意义的。
    Header() Header
    // WriteHeader该方法发送HTTP回复的头域和状态码。
    // 如果没有被显式调用，第一次调用Write时会触发隐式调用WriteHeader(http
    // WriterHeader的显式调用主要用于发送错误码。
    WriteHeader(int)
    // Write向连接中写入作为HTTP的一部分回复的数据。
    // 如果被调用时还未调用WriteHeader，本方法会先调用WriteHeader(http.St
    // 如果Header中没有"Content-Type"键，
    // 本方法会使用包函数DetectContentType检查数据的前512字节，将返回值作为
    Write([]byte) (int, error)
}
```

ResponseWriter接口被HTTP处理器用于构造HTTP回复。

type Flusher

```
type Flusher interface {
    // Flush将缓冲中的所有数据发送到客户端
    Flush()
}
```

HTTP处理器ResponseWriter接口参数的下层如果实现了Flusher接口，可以让HTTP处理器将缓冲中的数据发送到客户端。

注意：即使ResponseWriter接口的下层支持Flush方法，如果客户端是通过HTTP代理连接的，缓冲中的数据也可能直到回复完毕才被传输到客户端。

type CloseNotifier

```
type CloseNotifier interface {
    // CloseNotify返回一个通道，该通道会在客户端连接丢失时接收到唯一的值
    CloseNotify() <-chan bool
}
```

HTTP处理器ResponseWriter接口参数的下层如果实现了CloseNotifier接口，可以让用户检测下层的连接是否停止。如果客户端在回复准备好之前关闭了连接，该机制可以用于取消服务端耗时较长的操作。

type Hijacker

```
type Hijacker interface {
    // Hijack让调用者接管连接，返回连接和关联到该连接的一个缓冲读写器。
    // 调用本方法后，HTTP服务端将不再对连接进行任何操作，
    // 调用者有责任管理、关闭返回的连接。
    Hijack() (net.Conn, *bufio.ReadWriter, error)
}
```

HTTP处理器ResponseWriter接口参数的下层如果实现了Hijacker接口，可以让HTTP处理器接管该连接。

Example

```
http.HandleFunc("/hijack", func(w http.ResponseWriter, r *http.Requ
    hj, ok := w.(http.Hijacker)
    if !ok {
        http.Error(w, "webserver doesn't support hijacking", http.S
        return
    }
    conn, bufrw, err := hj.Hijack()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    // Don't forget to close the connection:
    defer conn.Close()
    bufrw.WriteString("Now we're speaking raw TCP. Say hi: ")
    bufrw.Flush()
    s, err := bufrw.ReadString('\n')
    if err != nil {
        log.Printf("error reading string: %v", err)
        return
    }
    fmt.Fprintf(bufrw, "You said: %q\nBye.\n", s)
    bufrw.Flush()
})
```

type RoundTripper

```
type RoundTripper interface {
    // RoundTrip执行单次HTTP事务，接收并发挥请求req的回复。
    // RoundTrip不应试图解析/修改得到的回复。
    // 尤其要注意，只要RoundTrip获得了一个回复，不管该回复的HTTP状态码如何，
    // 它必须将返回值err设置为nil。
    // 非nil的返回值err应该留给获取回复失败的情况。
    // 类似的，RoundTrip不能试图管理高层次的细节，如重定向、认证、cookie。
    //
    // 除了从请求的主体读取并关闭主体之外，RoundTrip不应修改请求，包括（请求
    // RoundTrip函数接收的请求的URL和Header字段可以保证是（被）初始化了的。
    RoundTrip(*Request) (*Response, error)
}
```

RoundTripper接口是具有执行单次HTTP事务的能力（接收指定请求的回复）的接口。

RoundTripper接口的类型必须可以安全的被多线程同时使用。

type Transport

```
type Transport struct {
    // Proxy指定一个对给定请求返回代理的函数。
    // 如果该函数返回了非nil的错误值，请求的执行就会中断并返回该错误。
    // 如果Proxy为nil或返回nil的*URL置，将不使用代理。
    Proxy func(*Request) (*url.URL, error)
    // Dial指定创建TCP连接的拨号函数。如果Dial为nil，会使用net.Dial。
    Dial func(network, addr string) (net.Conn, error)
    // TLSClientConfig指定用于tls.Client的TLS配置信息。
    // 如果该字段为nil，会使用默认的配置信息。
    TLSClientConfig *tls.Config
    // TLSHandshakeTimeout指定等待TLS握手完成的最长时间。零值表示不设置超时
    TLSHandshakeTimeout time.Duration
    // 如果DisableKeepAlives为真，会禁止不同HTTP请求之间TCP连接的重用。
    DisableKeepAlives bool
    // 如果DisableCompression为真，会禁止Transport在请求中没有Accept-En
    // 主动添加"Accept-Encoding: gzip"头，以获取压缩数据。
    // 如果Transport自己请求gzip并得到了压缩后的回复，它会主动解压缩回复的主
    // 但如果用户显式的请求gzip压缩数据，Transport是不会主动解压缩的。
    DisableCompression bool
    // 如果MaxIdleConnsPerHost!=0，会控制每个主机下的最大闲置连接。
    // 如果MaxIdleConnsPerHost==0，会使用DefaultMaxIdleConnsPerHost。
    MaxIdleConnsPerHost int
    // ResponseHeaderTimeout指定在发送完请求（包括其可能的主体）之后，
    // 等待接收服务端的回复的头域的最大时间。零值表示不设置超时。
    // 该时间不包括获取回复主体的时间。
    ResponseHeaderTimeout time.Duration
    // 内含隐藏或非导出字段
}
```

Transport类型实现了RoundTripper接口，支持http、https和http/https代理。Transport类型可以缓存连接以在未来重用。

```
var DefaultTransport RoundTripper = &Transport{
    Proxy: ProxyFromEnvironment,
    Dial: (&net.Dialer{
        Timeout:   30 * time.Second,
        KeepAlive: 30 * time.Second,
    }).Dial,
    TLSHandshakeTimeout: 10 * time.Second,
}
```

DefaultTransport是被包变量DefaultClient使用的默认RoundTripper接口。它会根据需要创建网络连接，并缓存以便在之后的请求中重用这些连接。它使用环境变量$HTTP_PROXY和$NO_PROXY（或$http_proxy和$no_proxy）指定的HTTP代理。

func (*Transport) RegisterProtocol

```
func (t *Transport) RegisterProtocol(scheme string, rt RoundTripper
```

RegisterProtocol注册一个新的名为scheme的协议。t会将使用scheme协议的请求转交给rt。rt有责任模拟HTTP请求的语义。

RegisterProtocol可以被其他包用于提供"ftp"或"file"等协议的实现。

func (*Transport) RoundTrip

```
func (t *Transport) RoundTrip(req *Request) (resp *Response, err er
```

RoundTrip方法实现了RoundTripper接口。

高层次的HTTP客户端支持（如管理cookie和重定向）请参见Get、Post等函数和Client类型。

func (*Transport) CloseIdleConnections

```
func (t *Transport) CloseIdleConnections()
```

CloseIdleConnections关闭所有之前的请求建立但目前处于闲置状态的连接。本方法不会中断正在使用的连接。

func (*Transport) CancelRequest

```
func (t *Transport) CancelRequest(req *Request)
```

CancelRequest通过关闭请求所在的连接取消一个执行中的请求。

type Client

```
type Client struct {
    // Transport指定执行独立、单次HTTP请求的机制。
    // 如果Transport为nil，则使用DefaultTransport。
    Transport RoundTripper
    // CheckRedirect指定处理重定向的策略。
    // 如果CheckRedirect不为nil，客户端会在执行重定向之前调用本函数字段。
    // 参数req和via是将要执行的请求和已经执行的请求（切片，越新的请求越靠后）
    // 如果CheckRedirect返回一个错误，本类型的Get方法不会发送请求req，
    // 而是返回之前得到的最后一个回复和该错误。（包装进url.Error类型里）
    //
    // 如果CheckRedirect为nil，会采用默认策略：连续10此请求后停止。
    CheckRedirect func(req *Request, via []*Request) error
    // Jar指定cookie管理器。
    // 如果Jar为nil，请求中不会发送cookie，回复中的cookie会被忽略。
    Jar CookieJar
    // Timeout指定本类型的值执行请求的时间限制。
    // 该超时限制包括连接时间、重定向和读取回复主体的时间。
    // 计时器会在Head、Get、Post或Do方法返回后继续运作并在超时后中断回复主体
    //
    // Timeout为零值表示不设置超时。
    //
    // Client实例的Transport字段必须支持CancelRequest方法，
    // 否则Client会在试图用Head、Get、Post或Do方法执行请求时返回错误。
    // 本类型的Transport字段默认值（DefaultTransport）支持CancelRequest
    Timeout time.Duration
}
```

Client类型代表HTTP客户端。它的零值（DefaultClient）是一个可用的使用DefaultTransport的客户端。

Client的Transport字段一般会含有内部状态（缓存TCP连接），因此Client类型值应尽量被重用而不是每次需要都创建新的。Client类型值可以安全的被多个go程同时使用。

Client类型的层次比RoundTripper接口（如Transport）高，还会管理HTTP的cookie和重定向等细节。

func (*Client) Do

```
func (c *Client) Do(req *Request) (resp *Response, err error)
```

Do方法发送请求，返回HTTP回复。它会遵守客户端c设置的策略（如重定向、cookie、认证）。

如果客户端的策略（如重定向）返回错误或存在HTTP协议错误时，本方法将返回该错误；如果回应的状态码不是2xx，本方法并不会返回错误。

如果返回值err为nil，resp.Body总是非nil的，调用者应该在读取完resp.Body后关闭它。如果返回值resp的主体未关闭，c下层的RoundTripper接口（一般为Transport类型）可能无法重用resp主体下层保持的TCP连接去执行之后的请求。

请求的主体，如果非nil，会在执行后被c.Transport关闭，即使出现错误。

一般应使用Get、Post或PostForm方法代替Do方法。

func (*Client) Head

```
func (c *Client) Head(url string) (resp *Response, err error)
```

Head向指定的URL发出一个HEAD请求，如果回应的状态码如下，Head会在调用c.CheckRedirect后执行重定向：

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
```

func (*Client) Get

```
func (c *Client) Get(url string) (resp *Response, err error)
```

Get向指定的URL发出一个GET请求，如果回应的状态码如下，Get会在调用c.CheckRedirect后执行重定向：

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
```

如果c.CheckRedirect执行失败或存在HTTP协议错误时，本方法将返回该错误；如果回应的状态码不是2xx，本方法并不会返回错误。如果返回值err为nil，resp.Body总是非nil的，调用者应该在读取完resp.Body后关闭它。

func (*Client) Post

```
func (c *Client) Post(url string, bodyType string, body io.Reader)
```

Post向指定的URL发出一个POST请求。bodyType为POST数据的类型， body为POST数据，作为请求的主体。如果参数body实现了io.Closer接口，它会在发送请求后被关闭。调用者有责任在读取完返回值resp的主体后关闭它。

func (*Client) PostForm

```
func (c *Client) PostForm(url string, data url.Values) (resp *Respo
```

PostForm向指定的URL发出一个POST请求，url.Values类型的data会被编码为请求的主体。POST数据的类型一般会设为"application/x-www-form-urlencoded"。如果返回值err为nil，resp.Body总是非nil的，调用者应该在读取完resp.Body后关闭它。

type Handler

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

实现了Handler接口的对象可以注册到HTTP服务端，为特定的路径及其子树提供服务。

ServeHTTP应该将回复的头域和数据写入ResponseWriter接口然后返回。返回标志着该请求已经结束，HTTP服务端可以转移向该连接上的下一个请求。

func NotFoundHandler

```
func NotFoundHandler() Handler
```

NotFoundHandler返回一个简单的请求处理器，该处理器会对每个请求都回复"404 page not found"。

func RedirectHandler

```
func RedirectHandler(url string, code int) Handler
```

RedirectHandler返回一个请求处理器，该处理器会对每个请求都使用状态码code重定向到网址url。

func TimeoutHandler

```
func TimeoutHandler(h Handler, dt time.Duration, msg string) Handle
```

TimeoutHandler返回一个采用指定时间限制的请求处理器。

返回的Handler会调用h.ServeHTTP去处理每个请求，但如果某一次调用耗时超过了时间限制，该处理器会回复请求状态码503 Service Unavailable，并将msg作为回复的主体（如果msg为空字符串，将发送一个合理的默认信息）。在超时后，h对它的ResponseWriter接口参数的写入操作会返回ErrHandlerTimeout。

func StripPrefix

```
func StripPrefix(prefix string, h Handler) Handler
```

StripPrefix返回一个处理器，该处理器会将请求的URL.Path字段中给定前缀prefix去除后再交由h处理。StripPrefix会向URL.Path字段中没有给定前缀的请求回复404 page not found。

Example

```
// To serve a directory on disk (/tmp) under an alternate URL
// path (/tmpfiles/), use StripPrefix to modify the request
// URL's path before the FileServer sees it:
http.Handle("/tmpfiles/", http.StripPrefix("/tmpfiles/", http.FileS
```

type HandlerFunc

```
type HandlerFunc func(ResponseWriter, *Request)
```

HandlerFunc type是一个适配器，通过类型转换让我们可以将普通的函数作为HTTP处理器使用。如果f是一个具有适当签名的函数，HandlerFunc(f)通过调用f实现了Handler接口。

func (HandlerFunc) ServeHTTP

```
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request)
```

ServeHTTP方法会调用f(w, r)

type ServeMux

```
type ServeMux struct {
    // 内含隐藏或非导出字段
}
```

ServeMux类型是HTTP请求的多路转接器。它会将每一个接收的请求的URL与一个注册模式的列表进行匹配，并调用和URL最匹配的模式的处理器。

模式是固定的、由根开始的路径，如"/favicon.ico"，或由根开始的子树，如"/images/"（注意结尾的斜杠）。较长的模式优先于较短的模式，因此如果模式"/images/"和"/images/thumbnails/"都注册了处理器，后一个处理器会用于路径以"/images/thumbnails/"开始的请求，前一个处理器会接收到其余的路径在"/images/"子树下的请求。

注意，因为以斜杠结尾的模式代表一个由根开始的子树，模式"/"会匹配所有的未被其他注册的模式匹配的路径，而不仅仅是路径"/"。

模式也能（可选地）以主机名开始，表示只匹配该主机上的路径。指定主机的模式优先于一般的模式，因此一个注册了两个模式"/codesearch"和"codesearch.google.com/"的处理器不会接管目标为"http://www.google.com/"的请求。

ServeMux还会注意到请求的URL路径的无害化，将任何路径中包含"."或".."元素的请求重定向到等价的没有这两种元素的URL。（参见path.Clean函数）

func NewServeMux

```
func NewServeMux() *ServeMux
```

NewServeMux创建并返回一个新的*ServeMux

func (*ServeMux) Handle

```
func (mux *ServeMux) Handle(pattern string, handler Handler)
```

Handle注册HTTP处理器handler和对应的模式pattern。如果该模式已经注册有一个处理器，Handle会panic。

Example

```
mux := http.NewServeMux()
mux.Handle("/api/", apiHandler{})
mux.HandleFunc("/", func(w http.ResponseWriter, req *http.Request)
    // The "/" pattern matches everything, so we need to check
    // that we're at the root here.
    if req.URL.Path != "/" {
        http.NotFound(w, req)
        return
    }
    fmt.Fprintf(w, "Welcome to the home page!")
})
```

func (*ServeMux) HandleFunc

```
func (mux *ServeMux) HandleFunc(pattern string, handler func(Respon
```

HandleFunc注册一个处理器函数handler和对应的模式pattern。

func (*ServeMux) Handler

```
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern strin
```

Handler根据r.Method、r.Host和r.URL.Path等数据，返回将用于处理该请求的HTTP处理器。它总是返回一个非nil的处理器。如果路径不是它的规范格式，将返回内建的用于重定向到等价的规范路径的处理器。

Handler也会返回匹配该请求的的已注册模式；在内建重定向处理器的情况下，pattern会在重定向后进行匹配。如果没有已注册模式可以应用于该请求，本方法将返回一个内建的"404 page not found"处理器和一个空字符串模式。

func (*ServeMux) ServeHTTP

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
```

ServeHTTP将请求派遣到与请求的URL最匹配的模式对应的处理器。

type Server

```
type Server struct {
    Addr           string        // 监听的TCP地址，如果为空字符串会使用"
    Handler        Handler       // 调用的处理器，如为nil会调用http.Def
    ReadTimeout    time.Duration // 请求的读取操作在超时前的最大持续时间
    WriteTimeout   time.Duration // 回复的写入操作在超时前的最大持续时间
    MaxHeaderBytes int           // 请求的头域最大长度，如为0则用Defaul
    TLSConfig      *tls.Config   // 可选的TLS配置，用于ListenAndServe
    // TLSNextProto（可选地）指定一个函数来在一个NPN型协议升级出现时接管TLS
    // 映射的键为商谈的协议名；映射的值为函数，该函数的Handler参数应处理HTTP
    // 并且初始化Handler.ServeHTTP的*Request参数的TLS和RemoteAddr字段（
    // 连接在函数返回时会自动关闭。
    TLSNextProto map[string]func(*Server, *tls.Conn, Handler)
    // ConnState字段指定一个可选的回调函数，该函数会在一个与客户端的连接改变
    // 参见ConnState类型和相关常数获取细节。
    ConnState func(net.Conn, ConnState)
    // ErrorLog指定一个可选的日志记录器，用于记录接收连接时的错误和处理器不正
    // 如果本字段为nil，日志会通过log包的标准日志记录器写入os.Stderr。
    ErrorLog *log.Logger
    // 内含隐藏或非导出字段
}
```

Server类型定义了运行HTTP服务端的参数。Server的零值是合法的配置。

func (*Server) SetKeepAlivesEnabled

```
func (s *Server) SetKeepAlivesEnabled(v bool)
```

SetKeepAlivesEnabled控制是否允许HTTP闲置连接重用（keep-alive）功能。默认该功能总是被启用的。只有资源非常紧张的环境或者服务端在关闭进程中时，才应该关闭该功能。

func (*Server) Serve

```
func (srv *Server) Serve(l net.Listener) error
```

Serve会接手监听器l收到的每一个连接，并为每一个连接创建一个新的服务go程。该go程会读取请求，然后调用srv.Handler回复请求。

func (*Server) ListenAndServe

```
func (srv *Server) ListenAndServe() error
```

ListenAndServe监听srv.Addr指定的TCP地址，并且会调用Serve方法接收到的连接。如果srv.Addr为空字符串，会使用":http"。

func (*Server) ListenAndServeTLS

```
func (srv *Server) ListenAndServeTLS(certFile, keyFile string) erro
```

ListenAndServeTLS监听srv.Addr确定的TCP地址，并且会调用Serve方法处理接收到的连接。必须提供证书文件和对应的私钥文件。如果证书是由权威机构签发的，certFile参数必须是顺序串联的服务端证书和CA证书。如果srv.Addr为空字符串，会使用":https"。

type File

```
type File interface {
    io.Closer
    io.Reader
    Readdir(count int) ([]os.FileInfo, error)
    Seek(offset int64, whence int) (int64, error)
    Stat() (os.FileInfo, error)
}
```

File是被FileSystem接口的Open方法返回的接口类型，可以被FileServer等函数用于文件访问服务。

该接口的方法的行为应该和*os.File类型的同名方法相同。

type FileSystem

```
type FileSystem interface {
    Open(name string) (File, error)
}
```

FileSystem接口实现了对一系列命名文件的访问。文件路径的分隔符为'/'，不管主机操作系统的惯例如何。

type Dir

```
type Dir string
```

Dir使用限制到指定目录树的本地文件系统实现了http.FileSystem接口。空Dir被视为"."，即代表当前目录。

func (Dir) Open

```
func (d Dir) Open(name string) (File, error)
```

func NewFileTransport

```
func NewFileTransport(fs FileSystem) RoundTripper
```

NewFileTransport返回一个RoundTripper接口，使用FileSystem接口fs提供文件访问服务。 返回的RoundTripper接口会忽略接收的请求的URL主机及其他绝大多数属性。

NewFileTransport函数的典型使用情况是给Transport类型的值注册"file"协议，如下所示：

```
t := &http.Transport{}
t.RegisterProtocol("file", http.NewFileTransport(http.Dir("/")))
c := &http.Client{Transport: t}
res, err := c.Get("file:///etc/passwd")
...
```

func FileServer

```
func FileServer(root FileSystem) Handler
```

FileServer返回一个使用FileSystem接口root提供文件访问服务的HTTP处理器。要使用操作系统的FileSystem接口实现，可使用http.Dir：

```
http.Handle("/", http.FileServer(http.Dir("/tmp")))
```

Example

```
// Simple static webserver:
log.Fatal(http.ListenAndServe(":8080", http.FileServer(http.Dir("/u
```

Example (StripPrefix)

```
// To serve a directory on disk (/tmp) under an alternate URL
// path (/tmpfiles/), use StripPrefix to modify the request
// URL's path before the FileServer sees it:
http.Handle("/tmpfiles/", http.StripPrefix("/tmpfiles/", http.FileS
```

func ProxyURL

```
func ProxyURL(fixedURL *url.URL) func(*Request) (*url.URL, error)
```

ProxyURL返回一个代理函数（用于Transport类型），该函数总是返回同一个URL。

func ProxyFromEnvironment

```
func ProxyFromEnvironment(req *Request) (*url.URL, error)
```

ProxyFromEnvironment使用环境变量$HTTP_PROXY和$NO_PROXY(或$http_proxy和$no_proxy)的配置返回用于req的代理。如果代理环境不合法将返回错误；如果环境未设定代理或者给定的request不应使用代理时，将返回(nil, nil)；如果req.URL.Host字段是"localhost"（可以有端口号，也可以没有），也会返回(nil, nil)。

func SetCookie

```
func SetCookie(w ResponseWriter, cookie *Cookie)
```

SetCookie在w的头域中添加Set-Cookie头，该HTTP头的值为cookie。

func Redirect

```
func Redirect(w ResponseWriter, r *Request, urlStr string, code int
```

Redirect回复请求一个重定向地址urlStr和状态码code。该重定向地址可以是相对于请求r的相对地址。

func NotFound

```
func NotFound(w ResponseWriter, r *Request)
```

NotFound回复请求404状态码（not found：目标未发现）。

func Error

```
func Error(w ResponseWriter, error string, code int)
```

Error使用指定的错误信息和状态码回复请求，将数据写入w。错误信息必须是明文。

func ServeContent

```
func ServeContent(w ResponseWriter, req *Request, name string, modt
```

ServeContent使用提供的ReadSeeker的内容回复请求。ServeContent比起io.Copy函数的主要优点，是可以处理范围类请求（只要一部分内容）、设置MIME类型，处理If-Modified-Since请求。

如果未设定回复的Content-Type头，本函数首先会尝试从name的文件扩展名推断数据类型；如果失败，会用读取content的第1块数据并提供给DetectContentType推断类型；之后会设置Content-Type头。参数name不会用于别的地方，甚至于它可以是空字符串，也永远不会发送到回复里。

如果modtime不是Time零值，函数会在回复的头域里设置Last-Modified头。如果请求的头域包含If-Modified-Since头，本函数会使用modtime参数来确定是否应该发送内容。如果调用者设置了w的ETag头，ServeContent会使用它处理包含If-Range头和If-None-Match头的请求。

参数content的Seek方法必须有效：函数使用Seek来确定它的大小。

注意：本包File接口和*os.File类型都实现了io.ReadSeeker接口。

func ServeFile

```
func ServeFile(w ResponseWriter, r *Request, name string)
```

ServeFile回复请求name指定的文件或者目录的内容。

func MaxBytesReader

```
func MaxBytesReader(w ResponseWriter, r io.ReadCloser, n int64) io
```

MaxBytesReader类似io.LimitReader，但它是用来限制接收到的请求的Body的大小的。不同于io.LimitReader，本函数返回一个ReadCloser，返回值的Read方法在读取的数据超过大小限制时会返回非EOF错误，其Close方法会关闭下层的io.ReadCloser接口r。

MaxBytesReader预防客户端因为意外或者蓄意发送的"大"请求，以避免尺寸过大的请求浪费服务端资源。

func Head

```
func Head(url string) (resp *Response, err error)
```

Head向指定的URL发出一个HEAD请求，如果回应的状态码如下，Head会在调用c.CheckRedirect后执行重定向：

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
```

Head是对包变量DefaultClient的Head方法的包装。

func Get

```
func Get(url string) (resp *Response, err error)
```

Get向指定的URL发出一个GET请求，如果回应的状态码如下，Get会在调用c.CheckRedirect后执行重定向：

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
```

如果c.CheckRedirect执行失败或存在HTTP协议错误时，本方法将返回该错误；如果回应的状态码不是2xx，本方法并不会返回错误。如果返回值err为nil，resp.Body总是非nil的，调用者应该在读取完resp.Body后关闭它。

Get是对包变量DefaultClient的Get方法的包装。

Example

```
res, err := http.Get("http://www.google.com/robots.txt")
if err != nil {
    log.Fatal(err)
}
robots, err := ioutil.ReadAll(res.Body)
res.Body.Close()
if err != nil {
    log.Fatal(err)
}
fmt.Printf("%s", robots)
```

func Post

```
func Post(url string, bodyType string, body io.Reader) (resp *Respo
```

Post向指定的URL发出一个POST请求。bodyType为POST数据的类型， body为POST数据，作为请求的主体。如果参数body实现了io.Closer接口，它会在发送请求后被关闭。调用者有责任在读取完返回值resp的主体后关闭它。

Post是对包变量DefaultClient的Post方法的包装。

func PostForm

```
func PostForm(url string, data url.Values) (resp *Response, err err
```

PostForm向指定的URL发出一个POST请求，url.Values类型的data会被编码为请求的主体。如果返回值err为nil，resp.Body总是非nil的，调用者应该在读取完resp.Body后关闭它。

PostForm是对包变量DefaultClient的PostForm方法的包装。

func Handle

```
func Handle(pattern string, handler Handler)
```

Handle注册HTTP处理器handler和对应的模式pattern（注册到DefaultServeMux）。如果该模式已经注册有一个处理器，Handle会panic。ServeMux的文档解释了模式的匹配机制。

func HandleFunc

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Reque
```

HandleFunc注册一个处理器函数handler和对应的模式pattern（注册到DefaultServeMux）。ServeMux的文档解释了模式的匹配机制。

func Serve

```
func Serve(l net.Listener, handler Handler) error
```

Serve会接手监听器l收到的每一个连接，并为每一个连接创建一个新的服务go程。该go程会读取请求，然后调用handler回复请求。handler参数一般会设为nil，此时会使用DefaultServeMux。

func ListenAndServe

```
func ListenAndServe(addr string, handler Handler) error
```

ListenAndServe监听TCP地址addr，并且会使用handler参数调用Serve函数处理接收到的连接。handler参数一般会设为nil，此时会使用DefaultServeMux。

一个简单的服务端例子：

```
package main
import (
    "io"
    "net/http"
    "log"
)
// hello world, the web server
func HelloServer(w http.ResponseWriter, req *http.Request) {
    io.WriteString(w, "hello, world!\n")
}
func main() {
    http.HandleFunc("/hello", HelloServer)
    err := http.ListenAndServe(":12345", nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

func ListenAndServeTLS

```
func ListenAndServeTLS(addr string, certFile string, keyFile string
```

ListenAndServeTLS函数和ListenAndServe函数的行为基本一致，除了它期望HTTPS连接之外。此外，必须提供证书文件和对应的私钥文件。如果证书是由权威机构签发的，certFile参数必须是顺序串联的服务端证书和CA证书。如果srv.Addr为空字符串，会使用":https"。

一个简单的服务端例子：

```
import (
    "log"
    "net/http"
)
func handler(w http.ResponseWriter, req *http.Request) {
    w.Header().Set("Content-Type", "text/plain")
    w.Write([]byte("This is an example server.\n"))
}
func main() {
    http.HandleFunc("/", handler)
    log.Printf("About to listen on 10443\. Go to https://127.0.0.1
    err := http.ListenAndServeTLS(":10443", "cert.pem", "key.pem",
    if err != nil {
        log.Fatal(err)
    }
}
```

程序员可以使用crypto/tls包的generate_cert.go文件来生成cert.pem和key.pem两个文件。

package cgi

```
import "net/http/cgi"
```

cgi包实现了CGI（Common Gateway Interface，公共网关协议），参见RFC 3875。

注意使用CGI意味着对每一个请求开始一个新的进程，这显然要比使用长期运行的服务程序要低效。本包主要是为了兼容现有的系统。

Index

func Request

```
func Request() (*http.Request, error)
```

返回一个当前环境下的HTTP请求。它假设当前程序执行在CGI环境下，成功返回的Request的Body是可读取数据的。（如果必要）

func RequestFromMap

```
func RequestFromMap(params map[string]string) (*http.Request, erro
```

使用CGI变量集params创建一个HTTP请求。返回的Request的Body是没有数据的。

func Serve

```
func Serve(handler http.Handler) error
```

在当前活跃CGI环境下执行handler；如当前无CGI环境，会返回错误。handler为nil时将使用http.DefaultServeMux。

type Handler

```
type Handler struct {
    Path string // CGI可执行文件的路径
    Root string // handler的根URI前缀，""代表"/"
    // Dir指定CGI程序的工作目录。
    // 如果Dir为""则使用Path的基目录；如果Path没有基目录则使用当前工作目录。
    Dir string
    Env        []string    // 额外设置的环境变量（如果有），格式为"key=va
    InheritEnv []string    // 从host继承的环境变量，只有"key"
    Logger     *log.Logger // 可选的logger接口切片，如为nil则使用log.Pr
    Args       []string    // 可选的传递给子进程的参数
    // 当CGI进程返回一个Location头，且其值以"/"开始时，
    // 本字段指定处理内部重定向的根部HTTP Handler。参见RFC 3875 § 6.3.2。
    // 一般会使用http.DefaultServeMux。
    // 如果为nil，返回一个本地URI路径的CGI回复会发送给客户端，不进行内部跳转。
    PathLocationHandler http.Handler
}
```

Handler在子进程中执行具有一个CGI环境的可执行程序。

func (*Handler) ServeHTTP

```
func (h *Handler) ServeHTTP(rw http.ResponseWriter, req *http.Reque
```

package cookiejar

```
import "net/http/cookiejar"
```

cookiejar包实现了保管在内存中的符合RFC 6265标准的http.CookieJar接口。

Index

type PublicSuffixList

```
type PublicSuffixList interface {
    // 返回域名的公共后缀。
    // TODO：域名的格式化应该由调用者还是接口方法负责还没有确定。
    PublicSuffix(domain string) string
    // 返回公共后缀列表的来源的说明，该说明一般应该包含时间戳和版本号。
    String() string
}
```

PublicSuffixList提供域名的公共后缀。例如：

```
- "example.com"的公共后缀是"com"
- "foo1.foo2.foo3.co.uk"的公共后缀是"co.uk"
- "bar.pvt.k12.ma.us"的公共后缀是"pvt.k12.ma.us"
```

PublicSuffixList接口的实现必须是并发安全的。一个总是返回""的实现是合法的，也可以通过测试；但却是不安全的：它允许HTTP服务端跨域名设置cookie。推荐实现：code.google.com/p/go.net/publicsuffix

type Options

```
type Options struct {
    // PublicSuffixList是公共后缀列表，用于决定HTTP服务端是否能给某域名设置
    // nil值合法的，也可以通过测试；但却是不安全的：它允许HTTP服务端跨域名设置
    PublicSuffixList PublicSuffixList
}
```

Options是创建新Jar是的选项。

type Jar

```
type Jar struct {
    // 内含隐藏或非导出字段
}
```

Jar类型实现了net/http包的http.CookieJar接口。

func New

```
func New(o *Options) (*Jar, error)
```

返回一个新的Jar，nil指针等价于Options零值的指针。

func (*Jar) Cookies

```
func (j *Jar) Cookies(u *url.URL) (cookies []*http.Cookie)
```

实现CookieJar接口的Cookies方法，如果URL协议不是HTTP/HTTPS会返回空切片。

func (*Jar) SetCookies

```
func (j *Jar) SetCookies(u *url.URL, cookies []*http.Cookie)
```

实现CookieJar接口的SetCookies方法，如果URL协议不是HTTP/HTTPS则不会有实际操作。

package fcgi

```
import "net/http/fcgi"
```

fcgi包实现了FastCGI协议。目前只支持响应器的角色。

协议定义的地址：http://www.fastcgi.com/drupal/node/6?q=node/22

Index

func Serve

```
func Serve(l net.Listener, handler http.Handler) error
```

Serve接受从监视器l传入的FastCGI连接，为每一个FastCGI连接创建一个新的go程。该go程读取请求然后调用handler回复请求。如果l是nil，Serve将从os.Stdin接受连接。如果handler是nil，将使用http.DefaultServeMux。

package httptest

```
import "net/http/httptest"
```

httptest包提供了HTTP测试的常用函数。

Index

Examples

Constants

```
const DefaultRemoteAddr = "1.2.3.4"
```

DefaultRemoteAddr是默认的远端地址。如果ResponseRecorder未显式的设置该属性，RemoteAddr方法就会返回该值。

type ResponseRecorder

```
type ResponseRecorder struct {
    Code      int           // HTTP回复的状态码
    HeaderMap http.Header   // HTTP回复的头域
    Body      *bytes.Buffer // 如非nil，会将Write方法写入的数据写入byte
    Flushed   bool
    // 内含隐藏或非导出字段
}
```

ResponseRecorder实现了http.ResponseWriter接口，它记录了其修改，用于之后的检查。

Example

```
handler := func(w http.ResponseWriter, r *http.Request) {
    http.Error(w, "something failed", http.StatusInternalServerErr
}
req, err := http.NewRequest("GET", "http://example.com/foo", nil)
if err != nil {
    log.Fatal(err)
}
w := httptest.NewRecorder()
handler(w, req)
fmt.Printf("%d - %s", w.Code, w.Body.String())
```

Output:

```
500 - something failed
```

func NewRecorder

```
func NewRecorder() *ResponseRecorder
```

NewRecorder返回一个初始化了的ResponseRecorder.

func (*ResponseRecorder) Header

```
func (rw *ResponseRecorder) Header() http.Header
```

Header返回回复的头域，即Header字段。

func (*ResponseRecorder) WriteHeader

```
func (rw *ResponseRecorder) WriteHeader(code int)
```

WriteHeader设置rw.Code。

func (*ResponseRecorder) Write

```
func (rw *ResponseRecorder) Write(buf []byte) (int, error)
```

Write总是成功，如果rw.Body非nil会把数据写入该字段。

func (*ResponseRecorder) Flush

```
func (rw *ResponseRecorder) Flush()
```

Flush将rw.Flushed设置为真。

type Server

```
type Server struct {
    URL      string // 格式为http://ipaddr:port，没有末尾斜杠的基地址
    Listener net.Listener
    // TLS是可选的TLS配置，在TLS开始后会填写为新的配置。
    // 如果在未启动的Server调用StartTLS方法前设置，已经存在的字段会拷贝进新
    TLS *tls.Config
    // Config可能会在调用Start/StartTLS方法之前调用NewUnstartedServer时
    Config *http.Server
    // 内含隐藏或非导出字段
}
```

Server是一个HTTP服务端，在本地环回接口的某个系统选择的端口监听，用于点对点HTTP测试。

Example

```
ts := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWrite
    fmt.Fprintln(w, "Hello, client")
}))
defer ts.Close()
res, err := http.Get(ts.URL)
if err != nil {
    log.Fatal(err)
}
greeting, err := ioutil.ReadAll(res.Body)
res.Body.Close()
if err != nil {
    log.Fatal(err)
}
fmt.Printf("%s", greeting)
```

Output:

```
Hello, client
```

func NewServer

```
func NewServer(handler http.Handler) *Server
```

NewServer返回一个新的、已启动的Server。调用者必须在用完时调用Close方法关闭它。

func NewTLSServer

```
func NewTLSServer(handler http.Handler) *Server
```

NewTLSServer返回一个新的、使用TLS的、已启动的Server。调用者必须在用完时调用Close方法关闭它。

func NewUnstartedServer

```
func NewUnstartedServer(handler http.Handler) *Server
```

NewUnstartedServer返回一个新的、未启动的Server。在修改其配置后，调用者应该调用Start或StartTLS启动它；调在用完时用者必须调用Close方法关闭它。

func (*Server) Start

```
func (s *Server) Start()
```

Start启动NewUnstartedServer返回的服务端。

func (*Server) StartTLS

```
func (s *Server) StartTLS()
```

StartTLS启动NewUnstartedServer函数返回的服务端的TLS监听。

func (*Server) CloseClientConnections

```
func (s *Server) CloseClientConnections()
```

CloseClientConnections关闭当前任何与该服务端建立的HTTP连接。

func (*Server) Close

```
func (s *Server) Close()
```

Close关闭服务端，并阻塞直到所有该服务端未完成的请求都结束为止。

package httputil

`import "net/http/httputil"`

httputil包提供了HTTP公用函数，是对net/http包的更常见函数的补充。

Index

Variables

```
var (
    ErrPersistEOF = &http.ProtocolError{ErrorString: "persistent co
    ErrClosed     = &http.ProtocolError{ErrorString: "connection cl
    ErrPipeline   = &http.ProtocolError{ErrorString: "pipeline erro
)
```

```
var ErrLineTooLong = errors.New("header line too long")
```

func DumpRequest

```
func DumpRequest(req *http.Request, body bool) (dump []byte, err er
```

DumpRequest返回req的和被服务端接收到时一样的有线表示，可选地包括请求的主体，用于debug。本函数在语义上是无操作的，但为了转储请求主体，他会读取主体的数据到内存中，并将req.Body修改为指向内存中的拷贝。req的字段的使用细节请参见http.Request的文档。

func DumpRequestOut

```
func DumpRequestOut(req *http.Request, body bool) ([]byte, error)
```

DumpRequestOut类似DumpRequest，但会包括标准http.Transport类型添加的头域，如User-Agent。

func DumpResponse

```
func DumpResponse(resp *http.Response, body bool) (dump []byte, err
```

DumpResponse类似DumpRequest，但转储的是一个回复。

func NewChunkedReader

```
func NewChunkedReader(r io.Reader) io.Reader
```

NewChunkedReader返回一个io.Reader。返回值的Read方法会将从r读取的采用HTTP "chunked"传输编码的数据翻译之后返回。当读取到最后的零长chunk时，返回值的Read会返回io.EOF。

NewChunkedReader在正常的应用中是不需要的，http包在读取回复主体时会自动将"chunked"编码进行解码。

func NewChunkedWriter

```
func NewChunkedWriter(w io.Writer) io.WriteCloser
```

NewChunkedWriter返回一个io.Writer。返回值的Write方法会将写入的数据编码为HTTP "chunked"传输编码格式后再写入w。其Close方法会将最后的零长chunk写入w，标注数据流的结尾。

正常的应用中不需要NewChunkedWriter，http包会在处理器未设置Content-Length头时主动进行chunked编码。在处理器内部使用本函数会导致双重chunked或者有Content-Length头的chunked，两个都是错误的。

type ClientConn

```
type ClientConn struct {
    // 内含隐藏或非导出字段
}
```

ClientConn类型会在尊重HTTP keepalive逻辑的前提下，在下层的连接上发送请求和接收回复的头域。ClientConn类型支持通过Hijack方法劫持下层连接，取回对下层连接的控制权，按照调用者的预期处理该连接。

ClientConn是旧的、低层次的。应用程序应使用net/http包的Client类型和Transport类型代替它。

func NewClientConn

```
func NewClientConn(c net.Conn, r *bufio.Reader) *ClientConn
```

NewClientConn返回一个对c进行读写的ClientConn。如果r不是nil，它是从c读取时使用的缓冲。

func NewProxyClientConn

```
func NewProxyClientConn(c net.Conn, r *bufio.Reader) *ClientConn
```

NewProxyClientConn类似NewClientConn，但使用Request.WriteProxy方法将请求写入c。

func (*ClientConn) Pending

```
func (cc *ClientConn) Pending() int
```

Pending返回该连接上已发送但还未接收到回复的请求的数量。

func (*ClientConn) Write

```
func (cc *ClientConn) Write(req *http.Request) (err error)
```

Write写入一个请求。如果该连接已经在HTTP keepalive逻辑上关闭了（表示该连接不会再发送新的请求）返回ErrPersistEOF。如果req.Close设置为真，keepalive连接会在这次请求后逻辑上关闭，并通知远端的服务器。ErrUnexpectedEOF表示远端关闭了下层的TCP连接，这一般被视为优雅的（正常的）关闭。

func (*ClientConn) Read

```
func (cc *ClientConn) Read(req *http.Request) (resp *http.Response,
```

Read读取下一个回复。合法的回复可能会和ErrPersistEOF一起返回，这表示远端要求该请求是最后一个被服务的请求。Read可以和Write同时调用，但不能和另一个Read同时调用。

func (*ClientConn) Do

```
func (cc *ClientConn) Do(req *http.Request) (resp *http.Response, e
```

Do是一个便利方法，它会写入一个请求，并读取一个回复。（能不能保证二者对应？不知道）

func (*ClientConn) Hijack

```
func (cc *ClientConn) Hijack() (c net.Conn, r *bufio.Reader)
```

Hijack拆开ClientConn返回下层的连接和读取侧的缓冲，其中可能有部分剩余的数据。Hijack可以在调用者自身或者其Read方法发出keepalive逻辑的终止信号之前调用。调用者不应在Write或Read执行过程中调用Hijack。

func (*ClientConn) Close

```
func (cc *ClientConn) Close() error
```

Close调用Hijack，然后关闭下层的连接。

type ServerConn

```
type ServerConn struct {
    // 内含隐藏或非导出字段
}
```

ServerConn类型在下层的连接上接收请求和发送回复，直到HTTP keepalive逻辑结束。ServerConn类型支持通过Hijack方法劫持下层连接，取回对下层连接的控制权，按照调用者的预期处理该连接。ServerConn支持管道内套，例如，请求可以不和回复的发送同步的读取（但仍按照相同的顺序）。

ServerConn是旧的、低层次的。应用程序应使用net/http包的Server类型代替它。

func NewServerConn

```
func NewServerConn(c net.Conn, r *bufio.Reader) *ServerConn
```

NewServerConn返回一个新的从c读写的ServerConn。如果r补位nil，它将作为从c读取时的缓冲。

func (*ServerConn) Pending

```
func (sc *ServerConn) Pending() int
```

Pending返回该连接上已接收到但还未回复的请求的数量。

func (*ServerConn) Read

```
func (sc *ServerConn) Read() (req *http.Request, err error)
```

Read读取下一个请求。如果它优雅的决定不会再有更多的请求（例如，在HTTP/1.0连接的第一个请求之后，或者HTTP/1.1的一个具有" Connection:close "头的请求之后），会返回ErrPersistEOF。

func (*ServerConn) Write

```
func (sc *ServerConn) Write(req *http.Request, resp *http.Response)
```

Write写入req的回复resp。如要优雅的关闭该连接，可以设置resp.Close字段为真。Write方法应该尽量执行（以回复尽量多的请求），直到Write本身返回错误，而不应考虑读取侧返回的任何错误。

func (*ServerConn) Hijack

```
func (sc *ServerConn) Hijack() (c net.Conn, r *bufio.Reader)
```

Hijack拆开ServerConn返回下层的连接和读取侧的缓冲，其中可能有部分剩余的数据。Hijack可以在调用者自身或者其Read方法发出keepalive逻辑的终止信号之前调用。调用者不应在Write或Read执行过程中调用Hijack。

func (*ServerConn) Close

```
func (sc *ServerConn) Close() error
```

Close调用Hijack，然后关闭下层的连接。

type ReverseProxy

```
type ReverseProxy struct {
    // Director必须是将请求修改为新的请求的函数。
    // 修改后的请求会使用Transport发送，得到的回复会不经修改的返回给客户端。
    Director func(*http.Request)
    // Transport用于执行代理请求。
    // 如果本字段为nil，会使用http.DefaultTransport。
    Transport http.RoundTripper
    // FlushInterval指定拷贝回复的主体时将数据刷新给客户端的时间间隔。
    // 如果本字段为零值，不会进行周期的刷新。（拷贝完回复主体后再刷新）
    FlushInterval time.Duration
}
```

ReverseProxy是一个HTTP处理器，它接收一个请求，发送给另一个服务端，将回复转发给客户端。

func NewSingleHostReverseProxy

```
func NewSingleHostReverseProxy(target *url.URL) *ReverseProxy
```

NewSingleHostReverseProxy返回一个新的ReverseProxy。返回值会将请求的URL重写为target参数提供的协议、主机和基路径。如果target参数的Path字段为"/base"，接收到的请求的URL.Path为"/dir"，修改后的请求的URL.Path将会是"/base/dir"。

func (*ReverseProxy) ServeHTTP

```
func (p *ReverseProxy) ServeHTTP(rw http.ResponseWriter, req *http
```

package pprof

`import "net/http/pprof"`

pprof包通过它的HTTP服务端提供pprof可视化工具期望格式的运行时剖面文件数据服务。关于pprof的更多信息，参见http://code.google.com/p/google-perftools/。

本包一般只需导入获取其注册HTTP处理器的副作用。处理器的路径以/debug/pprof/开始。

要使用pprof，在你的程序里导入本包：

```
import _ "net/http/pprof"
```

如果你的应用还没有运行http服务器，你需要开始一个http服务器。添加"net/http"包和"log"包到你的导入列表，然后在main函数开始处添加如下代码：

```
go func() {
    log.Println(http.ListenAndServe("localhost:6060", nil))
}()
```

然后使用pprof工具查看堆剖面：

```
go tool pprof http://localhost:6060/debug/pprof/heap
```

或查看周期30秒的CPU剖面：

```
go tool pprof http://localhost:6060/debug/pprof/profile
```

或查看go程阻塞剖面：

```
go tool pprof http://localhost:6060/debug/pprof/block
```

要查看所有可用的剖面，在你的浏览器阅读http://localhost:6060/debug/pprof/。要学习这些运转的设施，访问：

```
http://blog.golang.org/2011/06/profiling-go-programs.html
```

Index

func Handler

```
func Handler(name string) http.Handler
```

Handler返回一个提供name指定的剖面文件的服务的HTTP处理器。

func Cmdline

```
func Cmdline(w http.ResponseWriter, r *http.Request)
```

Cmdline回应执行中程序的命令行，采用NUL字节分隔的参数。本包将它注册在/debug/pprof/cmdline。

func Index

```
func Index(w http.ResponseWriter, r *http.Request)
```

Index回复请求要求的pprof格式的剖面。例如，"/debug/pprof/heap"会回复"heap"剖面。Index会回复"/debug/pprof/" 请求一个列出所有可用的剖面的HTML页面。

func Profile

```
func Profile(w http.ResponseWriter, r *http.Request)
```

Profile回复pprof格式的CPU剖面。本包将它注册在/debug/pprof/profile。

func Symbol

```
func Symbol(w http.ResponseWriter, r *http.Request)
```

Symbol查看请求中列出的程序计数器，回复一个映射程序计数器到函数名的表格。本包将它注册在/debug/pprof/symbol。

package mail

```
import "net/mail"
```

mail包实现了邮件的解析。

本包大部分都遵守RFC 5322规定的语法，值得注意的区别是：

```
* 旧格式地址和嵌入远端信息的地址不会被解析
* 组地址不会被解析
* 不支持全部的间隔符（CFWS语法元素），如分属两行的地址
```

Index

Variables

```
var ErrHeaderNotPresent = errors.New("mail: header not in message")
```

type Address

```
type Address struct {
    Name    string // 固有名，可以为空
    Address string // user@domain
}
```

Address类型表示一个邮箱地址。

例如地址"Barry Gibbs <bg@example.com>"表示为Address{Name: "Barry Gibbs", Address: "bg@example.com"}

func ParseAddress

```
func ParseAddress(address string) (*Address, error)
```

解析单个的RFC 5322地址，例如"Barry Gibbs <bg@example.com>"。

func (*Address) String

```
func (a *Address) String() string
```

将a代表的地址表示为合法的RFC 5322地址字符串。如果Name字段包含非ASCII字符将根据RFC 2047转义。

func ParseAddressList

```
func ParseAddressList(list string) ([]*Address, error)
```

函数将list作为一串邮箱地址并解析返回。

type Header

```
type Header map[string][]string
```

Header代表邮件头域的多个键值对。

func (Header) AddressList

```
func (h Header) AddressList(key string) ([]*Address, error)
```

将键key对应的值（字符串）作为邮箱地址列表解析并返回。

func (Header) Date

```
func (h Header) Date() (time.Time, error)
```

解析头域Date项的值并返回。

func (Header) Get

```
func (h Header) Get(key string) string
```

返回键key对应的第一个值，如果没有对应值，将返回空字符串。

type Message

```
type Message struct {
    Header Header
    Body   io.Reader
}
```

Message代表一个解析后的邮件。

func ReadMessage

```
func ReadMessage(r io.Reader) (msg *Message, err error)
```

从r读取一个邮件，会解析邮件头域，消息主体可以从r/msg.Body中读取。

package rpc

```
import "net/rpc"
```

rpc包提供了通过网络或其他I/O连接对一个对象的导出方法的访问。服务端注册一个对象，使它作为一个服务被暴露，服务的名字是该对象的类型名。注册之后，对象的导出方法就可以被远程访问。服务端可以注册多个不同类型的对象（服务），但注册具有相同类型的多个对象是错误的。

只有满足如下标准的方法才能用于远程访问，其余方法会被忽略：

```
- 方法是导出的
- 方法有两个参数，都是导出类型或内建类型
- 方法的第二个参数是指针
- 方法只有一个error接口类型的返回值
```

事实上，方法必须看起来像这样：

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

其中T、T1和T2都能被encoding/gob包序列化。这些限制即使使用不同的编解码器也适用。（未来，对定制的编解码器可能会使用较宽松一点的限制）

方法的第一个参数代表调用者提供的参数；第二个参数代表返回给调用者的参数。方法的返回值，如果非nil，将被作为字符串回传，在客户端看来就和errors.New创建的一样。如果返回了错误，回复的参数将不会被发送给客户端。

服务端可能会单个连接上调用ServeConn管理请求。更典型地，它会创建一个网络监听器然后调用Accept；或者，对于HTTP监听器，调用HandleHTTP和http.Serve。

想要使用服务的客户端会创建一个连接，然后用该连接调用NewClient。

更方便的函数Dial（DialHTTP）会在一个原始的连接（或HTTP连接）上依次执行这两个步骤。

生成的Client类型值有两个方法，Call和Go，它们的参数为要调用的服务和方法、一个包含参数的指针、一个用于接收接个的指针。

Call方法会等待远端调用完成，而Go方法异步的发送调用请求并使用返回的Call结构体类型的Done通道字段传递完成信号。

除非设置了显式的编解码器，本包默认使用encoding/gob包来传输数据。

这是一个简单的例子。一个服务端想要导出Arith类型的一个对象：

```
package server
type Args struct {
    A, B int
}
type Quotient struct {
    Quo, Rem int
}
type Arith int
func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}
func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

服务端会调用（用于HTTP服务）：

```
arith := new(Arith)
rpc.Register(arith)
rpc.HandleHTTP()
l, e := net.Listen("tcp", ":1234")
if e != nil {
    log.Fatal("listen error:", e)
}
go http.Serve(l, nil)
```

此时，客户端可看到服务"Arith"及它的方法"Arith.Multiply"、"Arith.Divide"。要调用方法，客户端首先呼叫服务端：

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
```

然后，客户端可以执行远程调用：

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
    log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)
```

或：

```
// Asynchronous call
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args, quotient, nil)
replyCall := <-divCall.Done    // will be equal to divCall
// check errors, print, etc.
```

服务端的实现应为客户端提供简单、类型安全的包装。

Index

Constants

```
const (
    // HandleHTTP使用的默认值
    DefaultRPCPath   = "/_goRPC_"
    DefaultDebugPath = "/debug/rpc"
)
```

Variables

```
var DefaultServer = NewServer()
```

DefaultServer是*Server的默认实例，本包和Server方法同名的函数都是对其方法的封装。

```
var ErrShutdown = errors.New("connection is shut down")
```

type ServerError

```
type ServerError string
```

ServerError represents an error that has been returned from the remote side of the RPC connection.

func (ServerError) Error

```
func (e ServerError) Error() string
```

type Request

```
type Request struct {
    ServiceMethod string // 格式："Service.Method"
    Seq           uint64 // 由客户端选择的序列号
    // 内含隐藏或非导出字段
}
```

Request是每个RPC调用请求的头域。它是被内部使用的，这里的文档用于帮助debug，如分析网络拥堵时。

type Response

```
type Response struct {
    ServiceMethod string // 对应请求的同一字段
    Seq           uint64 // 对应请求的同一字段
    Error         string // 可能的错误
    // 内含隐藏或非导出字段
}
```

Response是每个RPC调用回复的头域。它是被内部使用的，这里的文档用于帮助debug，如分析网络拥堵时。

type ClientCodec

```
type ClientCodec interface {
    // 本方法必须能安全的被多个go程同时使用
    WriteRequest(*Request, interface{}) error
    ReadResponseHeader(*Response) error
    ReadResponseBody(interface{}) error
    Close() error
}
```

ClientCodec接口实现了RPC会话的客户端一侧RPC请求的写入和RPC回复的读取。客户端调用WriteRequest来写入请求到连接，然后成对调用ReadRsponseHeader和ReadResponseBody以读取回复。客户端在结束该连接的事务时调用Close方法。ReadResponseBody可以使用nil参数调用，以强制回复的主体被读取然后丢弃。

type ServerCodec

```
type ServerCodec interface {
    ReadRequestHeader(*Request) error
    ReadRequestBody(interface{}) error
    // 本方法必须能安全的被多个go程同时使用
    WriteResponse(*Response, interface{}) error
    Close() error
}
```

ServerCodec接口实现了RPC会话的服务端一侧RPC请求的读取和RPC回复的写入。服务端通过成对调用方法ReadRequestHeader和ReadRequestBody从连接读取请求，然后调用WriteResponse来写入回复。服务端在结束该连接的事务时调用Close方法。ReadRequestBody可以使用nil参数调用，以强制请求的主体被读取然后丢弃。

type Call

```
type Call struct {
    ServiceMethod string      // 调用的服务和方法的名称
    Args          interface{} // 函数的参数（下层为结构体指针）
    Reply         interface{} // 函数的回复（下层为结构体指针）
    Error         error       // 在调用结束后，保管错误的状态
    Done          chan *Call  // 对其的接收操作会阻塞，直到远程调用结束
}
```

Call类型代表一个执行中/执行完毕的RPC会话。

type Client

```
type Client struct {
    // 内含隐藏或非导出字段
}
```

Client类型代表RPC客户端。同一个客户端可能有多个未返回的调用，也可能被多个go程同时使用。

func NewClient

```
func NewClient(conn io.ReadWriteCloser) *Client
```

NewClient返回一个新的Client，以管理对连接另一端的服务的请求。它添加缓冲到连接的写入侧，以便将回复的头域和有效负载作为一个单元发送。

func NewClientWithCodec

```
func NewClientWithCodec(codec ClientCodec) *Client
```

NewClientWithCodec类似NewClient，但使用指定的编解码器，以编码请求主体和解码回复主体。

func Dial

```
func Dial(network, address string) (*Client, error)
```

Dial在指定的网络和地址与RPC服务端连接。

func DialHTTP

```
func DialHTTP(network, address string) (*Client, error)
```

DialHTTP在指定的网络和地址与在默认HTTP RPC路径监听的HTTP RPC服务端连接。

func DialHTTPPath

```
func DialHTTPPath(network, address, path string) (*Client, error)
```

DialHTTPPath在指定的网络、地址和路径与HTTP RPC服务端连接。

func (*Client) Call

```
func (client *Client) Call(serviceMethod string, args interface{},
```

Call调用指定的方法，等待调用返回，将结果写入reply，然后返回执行的错误状态。

func (*Client) Go

```
func (client *Client) Go(serviceMethod string, args interface{}, re
```

Go异步的调用函数。本方法Call结构体类型指针的返回值代表该次远程调用。通道类型的参数done会在本次调用完成时发出信号（通过返回本次Go方法的返回值）。如果done为nil，Go会申请一个新的通道（写入返回值的Done字段）；如果done非nil，done必须有缓冲，否则Go方法会故意崩溃。

func (*Client) Close

```
func (client *Client) Close() error
```

type Server

```
type Server struct {
    // 内含隐藏或非导出字段
}
```

Server代表RPC服务端。

func NewServer

```
func NewServer() *Server
```

NewServer创建并返回一个*Server。

func (*Server) Register

```
func (server *Server) Register(rcvr interface{}) error
```

Register在server注册并公布rcvr的方法集中满足如下要求的方法：

```
- 方法是导出的
- 方法有两个参数，都是导出类型或内建类型
- 方法的第二个参数是指针
- 方法只有一个error接口类型的返回值
```

如果rcvr不是一个导出类型的值，或者该类型没有满足要求的方法，Register会返回错误。Register也会使用log包将错误写入日志。客户端可以使用格式为"Type.Method"的字符串访问这些方法，其中Type是rcvr的具体类型。

func (*Server) RegisterName

```
func (server *Server) RegisterName(name string, rcvr interface{}) e
```

RegisterName类似Register，但使用提供的name代替rcvr的具体类型名作为服务名。

func (*Server) Accept

```
func (server *Server) Accept(lis net.Listener)
```

Accept接收监听器l获取的连接，然后服务每一个连接。Accept会阻塞，调用者应另开线程："go server.Accept(l)"

func (*Server) ServeConn

```
func (server *Server) ServeConn(conn io.ReadWriteCloser)
```

ServeConn在单个连接上执行server。ServeConn会阻塞，服务该连接直到客户端挂起。调用者一般应另开线程调用本函数："go server.ServeConn(conn)"。ServeConn在该连接使用gob（参见encoding/gob包）有线格式。要使用其他的编解码器，可调用ServeCodec方法。

func (*Server) ServeCodec

```
func (server *Server) ServeCodec(codec ServerCodec)
```

ServeCodec类似ServeConn，但使用指定的编解码器，以编码请求主体和解码回复主体。

func (*Server) ServeRequest

```
func (server *Server) ServeRequest(codec ServerCodec) error
```

ServeRequest类似ServeCodec，但异步的服务单个请求。它不会在调用结束后关闭codec。

func (*Server) ServeHTTP

```
func (server *Server) ServeHTTP(w http.ResponseWriter, req *http.Re
```

ServeHTTP实现了回应RPC请求的http.Handler接口。

func (*Server) HandleHTTP

```
func (server *Server) HandleHTTP(rpcPath, debugPath string)
```

HandleHTTP注册server的RPC信息HTTP处理器对应到rpcPath，注册server的debug信息HTTP处理器对应到debugPath。HandleHTTP会注册到http.DefaultServeMux。之后，仍需要调用http.Serve()，一般会另开线程："go http.Serve(l, nil)"

func Register

```
func Register(rcvr interface{}) error
```

Register在DefaultServer注册并公布rcvr的方法。

func RegisterName

```
func RegisterName(name string, rcvr interface{}) error
```

RegisterName函数类似Register函数，但使用提供的name代替rcvr的具体类型名作为服务名。

func Accept

```
func Accept(lis net.Listener)
```

Accept接收监听器l获取的连接，然后将每一个连接交给DefaultServer服务。Accept会阻塞，调用者应另开线程："go server.Accept(l)"

func ServeConn

```
func ServeConn(conn io.ReadWriteCloser)
```

ServeConn在单个连接上执行DefaultServer。ServeConn会阻塞，服务该连接直到客户端挂起。调用者一般应另开线程调用本函数："go ServeConn(conn)"。ServeConn在该连接使用gob（参见encoding/gob包）有线格式。要使用其他的编解码器，可调用ServeCodec方法。

func ServeCodec

```
func ServeCodec(codec ServerCodec)
```

ServeCodec类似ServeConn，但使用指定的编解码器，以编码请求主体和解码回复主体。

func ServeRequest

```
func ServeRequest(codec ServerCodec) error
```

ServeRequest类似ServeCodec，但异步的服务单个请求。它不会在调用结束后关闭codec。

func HandleHTTP

```
func HandleHTTP()
```

HandleHTTP函数注册DefaultServer的RPC信息HTTP处理器对应到DefaultRPCPath，和DefaultServer的debug处理器对应到DefaultDebugPath。HandleHTTP函数会注册到http.DefaultServeMux。之后，仍需要调用http.Serve()，一般会另开线程："go http.Serve(l, nil)"

package jsonrpc

```
import "net/rpc/jsonrpc"
```

jsonrpc包实现了JSON-RPC的ClientCodec和ServerCodec接口，可用于rpc包。

Index

func Dial

```
func Dial(network, address string) (*rpc.Client, error)
```

Dial在指定的网络和地址连接一个JSON-RPC服务端。

func NewClient

```
func NewClient(conn io.ReadWriteCloser) *rpc.Client
```

NewClient返回一个新的rpc.Client，以管理对连接另一端的服务的请求。

func NewClientCodec

```
func NewClientCodec(conn io.ReadWriteCloser) rpc.ClientCodec
```

NewClientCodec返回一个在连接上使用JSON-RPC的rpc.ClientCodec。

func NewServerCodec

```
func NewServerCodec(conn io.ReadWriteCloser) rpc.ServerCodec
```

NewServerCodec返回一个在连接上使用JSON-RPC的rpc. ServerCodec。

func ServeConn

```
func ServeConn(conn io.ReadWriteCloser)
```

ServeConn在单个连接上执行DefaultServer。ServeConn会阻塞，服务该连接直到客户端挂起。调用者一般应另开线程调用本函数："go serveConn(conn)"。ServeConn在该连接使用JSON编解码格式。

package smtp

```
import "net/smtp"
```

smtp包实现了简单邮件传输协议（SMTP），参见RFC 5321。同时本包还实现了如下扩展：

```
8BITMIME   RFC 1652
AUTH       RFC 2554
STARTTLS   RFC 3207
```

客户端可以自行管理其他的扩展。

Example

```
// Connect to the remote SMTP server.
c, err := smtp.Dial("mail.example.com:25")
if err != nil {
    log.Fatal(err)
}
// Set the sender and recipient first
if err := c.Mail("sender@example.org"); err != nil {
    log.Fatal(err)
}
if err := c.Rcpt("recipient@example.net"); err != nil {
    log.Fatal(err)
}
// Send the email body.
wc, err := c.Data()
if err != nil {
    log.Fatal(err)
}
_, err = fmt.Fprintf(wc, "This is the email body")
if err != nil {
    log.Fatal(err)
}
err = wc.Close()
if err != nil {
    log.Fatal(err)
}
// Send the QUIT command and close the connection.
err = c.Quit()
if err != nil {
    log.Fatal(err)
}
```

Index

Examples

type ServerInfo

```
type ServerInfo struct {
    Name string   // SMTP服务器的名字
    TLS  bool     // Name有合法的证书并使用TLS时为真
    Auth []string // 支持的认证机制
}
```

ServerInfo类型记录一个SMTP服务器的信息。

type Auth

```
type Auth interface {
    // 方法开始和服务端的认证。
    // 它返回认证协议的名字和可能有的应发送给服务端的包含初始认证信息的数据。
    // 如果返回值proto == "", 表示应跳过认证；
    // 如果返回一个非nil的错误，SMTP客户端应中断认证身份的尝试并关闭连接。
    Start(server *ServerInfo) (proto string, toServer []byte, err e
    // 方法继续认证过程。fromServer为服务端刚发送的数据。
    // 如果more为真，服务端会期望一个回复，回复内容应被Next返回，即toServer
    // 否则返回值toServer应为nil。
    // 如果返回一个非nil的错误，SMTP客户端应中断认证身份的尝试并关闭连接。
    Next(fromServer []byte, more bool) (toServer []byte, err error
}
```

Auth接口应被每一个SMTP认证机制实现。

func CRAMMD5Auth

```
func CRAMMD5Auth(username, secret string) Auth
```

返回一个实现了CRAM-MD5身份认证机制（参见RFC 2195）的Auth接口。返回的接口使用给出的用户名和密码，采用响应——回答机制与服务端进行身份认证。

func PlainAuth

```
func PlainAuth(identity, username, password, host string) Auth
```

返回一个实现了PLAIN身份认证机制（参见RFC 4616）的Auth接口。返回的接口使用给出的用户名和密码，通过TLS连接到主机认证，采用identity为身份管理和行动（通常应设identity为""，以便使用username为身份）。

Example

```
// Set up authentication information.
auth := smtp.PlainAuth("", "user@example.com", "password", "mail.ex
// Connect to the server, authenticate, set the sender and recipien
// and send the email all in one step.
to := []string{"recipient@example.net"}
msg := []byte("This is the email body.")
err := smtp.SendMail("mail.example.com:25", auth, "sender@example.c
if err != nil {
    log.Fatal(err)
}
```

type Client

```
type Client struct {
    // 代表被Client使用的textproto.Conn，它可以导出，以便使用者添加扩展。
    Text *textproto.Conn
    // 内含隐藏或非导出字段
}
```

Client代表一个连接到SMTP服务器的客户端。

func Dial

```
func Dial(addr string) (*Client, error)
```

Dial返回一个连接到地址为addr的SMTP服务器的*Client；addr必须包含端口号。

func NewClient

```
func NewClient(conn net.Conn, host string) (*Client, error)
```

NewClient使用已经存在的连接conn和作为服务器名的host（用于身份认证）来创建一个*Client。

func (*Client) Extension

```
func (c *Client) Extension(ext string) (bool, string)
```

Extension返回服务端是否支持某个扩展，扩展名是大小写不敏感的。如果扩展被支持，方法还会返回一个包含指定给该扩展的各个参数的字符串。

func (*Client) Hello

```
func (c *Client) Hello(localName string) error
```

Hello发送给服务端一个HELO或EHLO命令。本方法只有使用者需要控制使用的本地主机名时才应使用，否则程序会将本地主机名设为“localhost”，Hello方法只能在最开始调用。

func (*Client) Auth

```
func (c *Client) Auth(a Auth) error
```

Auth使用提供的认证机制进行认证。失败的认证会关闭该连接。只有服务端支持AUTH时，本方法才有效。（但是不支持时，调用会默默的成功）

func (*Client) Verify

```
func (c *Client) Verify(addr string) error
```

Verify检查一个邮箱地址在其服务器是否合法，如果合法会返回nil；但非nil的返回值并不代表不合法，因为许多服务器出于安全原因不支持这种查询。

func (*Client) StartTLS

```
func (c *Client) StartTLS(config *tls.Config) error
```

StartTLS方法发送STARTTLS命令，并将之后的所有数据往来加密。只有服务器附加了STARTTLS扩展，这个方法才有效。

func (*Client) Mail

```
func (c *Client) Mail(from string) error
```

Mail发送MAIL命令和邮箱地址from到服务器。如果服务端支持8BITMIME扩展，本方法会添加BODY=8BITMIME参数。方法初始化一次邮件传输，后应跟1到多个Rcpt方法的调用。

func (*Client) Rcpt

```
func (c *Client) Rcpt(to string) error
```

Rcpt发送RCPT命令和邮箱地址to到服务器。调用Rcpt方法之前必须调用了Mail方法，之后可以再一次调用Rcpt方法，也可以调用Data方法。

func (*Client) Data

```
func (c *Client) Data() (io.WriteCloser, error)
```

Data发送DATA指令到服务器并返回一个io.WriteCloser，用于写入邮件信息。调用者必须在调用用c的下一个方法之前关闭这个io.WriteCloser。方法必须在一次或多次Rcpt方法之后调用。

func (*Client) Reset

```
func (c *Client) Reset() error
```

Reset向服务端发送REST命令，中断当前的邮件传输。

func (*Client) Quit

```
func (c *Client) Quit() error
```

Quit发送QUIT命令并关闭到服务端的连接。

func (*Client) Close

```
func (c *Client) Close() error
```

Close关闭连接。

func SendMail

```
func SendMail(addr string, a Auth, from string, to []string, msg [
```

SendMail连接到addr指定的服务器；如果支持会开启TLS；如果支持会使用a认证身份；然后以from为邮件源地址发送邮件msg到目标地址to。（可以是多个目标地址：群发）

package textproto

```
import "net/textproto"
```

textproto实现了对基于文本的请求/回复协议的一般性支持，包括HTTP、NNTP和SMTP。

本包提供：

错误，代表服务端回复的错误码。Pipeline，以管理客户端中的管道化的请求/回复。Reader，读取数值回复码行，键值对形式的头域，一个作为后续行先导的空行，以及以只有一个"."的一行为结尾的整个文本块。Writer，写入点编码的文本。Conn，对Reader、Writer和Pipline的易用的包装，用于单个网络连接。

Index

type ProtocolError

```
type ProtocolError string
```

ProtocolError描述一个违反协议的错误，如不合法的回复或者挂起的连接。

func (ProtocolError) Error

```
func (p ProtocolError) Error() string
```

type Error

```
type Error struct {
    Code int
    Msg  string
}
```

Error代表一个服务端返回的数值状态码/错误码。

func (*Error) Error

```
func (e *Error) Error() string
```

func CanonicalMIMEHeaderKey

```
func CanonicalMIMEHeaderKey(s string) string
```

返回一个MIME头的键的规范格式。该标准会将首字母和所有"-"之后的字符改为大写，其余字母改为小写。举个例子，"accept-encoding"作为键的标准格式是"Accept-Encoding"。MIME头的键必须是ASCII码构成。

func TrimBytes

```
func TrimBytes(b []byte) []byte
```

去掉b前后的ASCII码空白（不去Unicode空白）

func TrimString

```
func TrimString(s string) string
```

去掉s前后的ASCII码空白（不去Unicode空白）

type MIMEHeader

```
type MIMEHeader map[string][]string
```

MIMEHeader代表一个MIME头，将键映射为值的集合。

func (MIMEHeader) Get

```
func (h MIMEHeader) Get(key string) string
```

Get方法返回键对应的值集的第一个值。如果键没有关联值，返回""。如要获得键对应的值集直接用map。

func (MIMEHeader) Set

```
func (h MIMEHeader) Set(key, value string)
```

Set方法将键对应的值集设置为只含有value一个值。没有就新建，有则删掉原有的值。

func (MIMEHeader) Add

```
func (h MIMEHeader) Add(key, value string)
```

Add方法向h中添加键值对，它会把新的值添加到键对应的值的集合里。

func (MIMEHeader) Del

```
func (h MIMEHeader) Del(key string)
```

Del方法删除键对应的值集。

type Reader

```
type Reader struct {
    R *bufio.Reader
    // 内含隐藏或非导出字段
}
```

Reader实现了从一个文本协议网络连接中方便的读取请求/回复的方法。

func NewReader

```
func NewReader(r *bufio.Reader) *Reader
```

NewReader返回一个从r读取数据的Reader。

func (*Reader) DotReader

```
func (r *Reader) DotReader() io.Reader
```

DotReader方法返回一个io.Reader，该接口自动解码r中读取的点编码块。注意该接口仅在下一次调用r的方法之前才有效。点编码是文本协议如SMTP用于文本块的通用框架。数据包含多个行，每行以"\r\n"结尾。数据本身以一个只含有一个点的一行".\r\n"来结尾。以点起始的行会添加额外的点，来避免看起来像是文本的结尾。

返回接口的Read方法会将行尾的"\r\n"修改为"\n"，去掉起头的转义点，并在底层读取到（并抛弃掉）表示文本结尾的行时停止解码并返回io.EOF错误。

func (*Reader) ReadLine

```
func (r *Reader) ReadLine() (string, error)
```

ReadLine方法从r读取单行，去掉最后的\r\n或\n。

func (*Reader) ReadLineBytes

```
func (r *Reader) ReadLineBytes() ([]byte, error)
```

ReadLineBytes类似ReadLine但返回[]byte切片。

func (*Reader) ReadContinuedLine

```
func (r *Reader) ReadContinuedLine() (string, error)
```

ReadContinuedLine从r中读取可能有后续的行，会将该行尾段的ASCII空白剔除，并将该行后面所有以空格或者tab起始的行视为其后续，后续部分会剔除行头部的空白，所有这些行包括第一行以单个空格连接起来返回。

举例如下：

```
Line 1
  continued...
Line 2
```

第一次调用ReadContinuedLine会返回"Line 1 continued..."，第二次会返回"Line 2"

只有空格的行不被视为有后续的行。

func (*Reader) ReadContinuedLineBytes

```
func (r *Reader) ReadContinuedLineBytes() ([]byte, error)
```

ReadContinuedLineBytes类似ReadContinuedLine但返回[]byte切片。

func (*Reader) ReadDotBytes

```
func (r *Reader) ReadDotBytes() ([]byte, error)
```

ReadDotBytes读取点编码文本返回解码后的数据，点编码详见DotReader方法。

func (*Reader) ReadDotLines

```
func (r *Reader) ReadDotLines() ([]string, error)
```

ReadDotLines方法读取一个点编码文本块并返回一个包含解码后各行的切片，各行最后的\r\n或\n去掉。

func (*Reader) ReadCodeLine

```
func (r *Reader) ReadCodeLine(expectCode int) (code int, message st
```

方法读取回复的状态码行，格式如下：

```
code message
```

状态码是3位数字，message进一步描述状态，例如：

```
220 plan9.bell-labs.com ESMTP
```

如果状态码字符串的前缀不匹配expectCode，方法返回错误&Error{code, message}。例如expectCode是31，则如果状态码不在区间[310, 319]内就会返回错误。如果回复是多行的则会返回错误。

如果expectCode <= 0，将不会检查状态码。

func (*Reader) ReadResponse

```
func (r *Reader) ReadResponse(expectCode int) (code int, message st
```

ReadResponse方法读取如下格式的多行回复：

```
code-message line 1
code-message line 2
...
code message line n
```

其中code是三位数的状态码。第一行以code和连字符开始，最后以同code后跟空格的行结束。返回值message每行以\n分隔。细节参见RFC 959(http://www.ietf.org/rfc/rfc959.txt)第36页。

如果状态码字符串的前缀不匹配expectCode，方法返回时err设为&Error{code, message}。例如expectCode是31，则如果状态码不在区间[310, 319]内就会返回错误。如果回复是多行的则会返回错误。

如果expectCode <= 0，将不会检查状态码。

func (*Reader) ReadMIMEHeader

```
func (r *Reader) ReadMIMEHeader() (MIMEHeader, error)
```

ReadMIMEHeader从r读取MIME风格的头域。该头域包含一系列可能有后续的键值行，以空行结束。返回的map映射CanonicalMIMEHeaderKey(key)到值的序列（顺序与输入相同）。

举例如下：

```
My-Key: Value 1
Long-Key: Even
       Longer Value
My-Key: Value 2
```

对此输入，ReadMIMEHeader返回：

```
map[string][]string{
    "My-Key": {"Value 1", "Value 2"},
    "Long-Key": {"Even Longer Value"},
}
```

type Writer

```
type Writer struct {
    W *bufio.Writer
    // 内含隐藏或非导出字段
}
```

Writer实现了方便的方法在一个文本协议网络连接中写入请求/回复。

func NewWriter

```
func NewWriter(w *bufio.Writer) *Writer
```

NewWriter函数返回一个底层写入w的Writer。

func (*Writer) DotWriter

```
func (w *Writer) DotWriter() io.WriteCloser
```

DotWriter方法返回一个io.WriteCloser，用于将点编码文本写入w。返回的接口会在必要时添加转义点，将行尾的\n替换为\r\n，并在关闭时添加最后的.\r\n行。调用者必须在下一次调用w的方法前关闭该接口。点编码文本格式参见Reader.DotReader方法。

func (*Writer) PrintfLine

```
func (w *Writer) PrintfLine(format string, args ...interface{}) er
```

PrintfLine方法将格式化的输出写入底层并在最后写入\r\n。

type Pipeline

```
type Pipeline struct {
    // 内含隐藏或非导出字段
}
```

Pipeline管理管道化的有序请求/回复序列。

为了使用Pipeline管理一个连接的多个客户端，每个客户端应像下面一样运行：

```
id := p.Next()       // 获取一个数字id
p.StartRequest(id)  // 等待轮到该id发送请求
«send request»
p.EndRequest(id)    // 通知Pipeline请求发送完毕
p.StartResponse(id) // 等待该id读取回复
«read response»
p.EndResponse(id)   // 通知Pipeline回复已经读取
```

一个管道化的服务器可以使用相同的调用来保证回复并行的生成并以正确的顺序写入。

func (*Pipeline) Next

```
func (p *Pipeline) Next() uint
```

返回下一对request/response的id。

func (*Pipeline) StartRequest

```
func (p *Pipeline) StartRequest(id uint)
```

阻塞程序，直到轮到给定id来发送（读取）request。

func (*Pipeline) StartResponse

```
func (p *Pipeline) StartResponse(id uint)
```

阻塞程序，直到轮到给定id来读取（发送）response。

func (*Pipeline) EndRequest

```
func (p *Pipeline) EndRequest(id uint)
```

通知p，给定id的request的操作已经结束了。

func (*Pipeline) EndResponse

```
func (p *Pipeline) EndResponse(id uint)
```

通知p，给定id的response的操作已经结束了。

type Conn

```
type Conn struct {
    Reader
    Writer
    Pipeline
    // 内含隐藏或非导出字段
}
```

Conn代表一个文本网络协议的连接。它包含一个Reader和一个Writer来管理读写，一个Pipeline来对连接中并行的请求进行排序。匿名嵌入的类型字段是Conn可以调用它们的方法。

func NewConn

```
func NewConn(conn io.ReadWriteCloser) *Conn
```

NewConn函数返回以I/O为底层的Conn。

func Dial

```
func Dial(network, addr string) (*Conn, error)
```

Dial函数使用net.Dial在给定网络上和给定地址建立网络连接，并返回用于该连接的Conn。

func (*Conn) Cmd

```
func (c *Conn) Cmd(format string, args ...interface{}) (id uint, er
```

Cmd方法用于在管道中等待轮到它执行，并发送命令。命令文本是用给定的format字符串和参数格式化生成的。并会在最后添加上\r\n。Cmd函数返回该命令的Pipeline id，用于StartResponse和EndResponse方法。

例如，一个客户端可以使用如下代码执行HELP命令并返回解码后的点编码文本：

```
id, err := c.Cmd("HELP")
if err != nil {
    return nil, err
}
c.StartResponse(id)
defer c.EndResponse(id)
if _, _, err = c.ReadCodeLine(110); err != nil {
    return nil, err
}
text, err := c.ReadDotBytes()
if err != nil {
    return nil, err
}
return c.ReadCodeLine(250)
```

func (*Conn) Close

```
func (c *Conn) Close() error
```

Close方法关闭连接。

Bugs

☞为了让调用者处理拒绝服务攻击，Reader接口应该允许他们设置和重设从连接读取的字节数。

package url

```
import "net/url"
```

url包解析URL并实现了查询的逸码，参见RFC 3986。

Index

Examples

func QueryEscape

```
func QueryEscape(s string) string
```

QueryEscape函数对s进行转码使之可以安全的用在URL查询里。

func QueryUnescape

```
func QueryUnescape(s string) (string, error)
```

QueryUnescape函数用于将QueryEscape转码的字符串还原。它会把%AB改为字节0xAB，将'+'改为' '。如果有某个%后面未跟两个十六进制数字，本函数会返回错误。

type Error

```
type Error struct {
    Op  string
    URL string
    Err error
}
```

Error会报告一个错误，以及导致该错误发生的URL和操作。

func (*Error) Error

```
func (e *Error) Error() string
```

type EscapeError

```
type EscapeError string
```

func (EscapeError) Error

```
func (e EscapeError) Error() string
```

type URL

```
type URL struct {
    Scheme   string
    Opaque   string    // 编码后的不透明数据
    User     *Userinfo // 用户名和密码信息
    Host     string    // host或host:port
    Path     string
    RawQuery string // 编码后的查询字符串，没有'?'
    Fragment string // 引用的片段（文档位置），没有'#'
}
```

URL类型代表一个解析后的URL（或者说，一个URL参照）。URL基本格式如下：

```
scheme://[userinfo@]host/path[?query][#fragment]
```

scheme后不是冒号加双斜线的URL被解释为如下格式：

```
scheme:opaque[?query][#fragment]
```

注意路径字段是以解码后的格式保存的，如/%47%6f%2f会变成/Go/。这导致我们无法确定Path字段中的斜线是来自原始URL还是解码前的%2f。除非一个客户端必须使用其他程序/函数来解析原始URL或者重构原始URL，这个区别并不重要。此时，HTTP服务端可以查询req.RequestURI，而HTTP客户端可以使用URL{Host: "example.com", Opaque: "//example.com/Go%2f"}代替{Host: "example.com", Path: "/Go/"}。

Example

```
u, err := url.Parse("http://bing.com/search?q=dotnet")
if err != nil {
    log.Fatal(err)
}
u.Scheme = "https"
u.Host = "google.com"
q := u.Query()
q.Set("q", "golang")
u.RawQuery = q.Encode()
fmt.Println(u)
```

Output:

```
https://google.com/search?q=golang
```

func Parse

```
func Parse(rawurl string) (url *URL, err error)
```

Parse函数解析rawurl为一个URL结构体，rawurl可以是绝对地址，也可以是相对地址。

func ParseRequestURI

```
func ParseRequestURI(rawurl string) (url *URL, err error)
```

ParseRequestURI函数解析rawurl为一个URL结构体，本函数会假设rawurl是在一个HTTP请求里，因此会假设该参数是一个绝对URL或者绝对路径，并会假设该URL没有#fragment后缀。（网页浏览器会在去掉该后缀后才将网址发送到网页服务器）

func (*URL) IsAbs

```
func (u *URL) IsAbs() bool
```

函数在URL是绝对URL时才返回真。

func (*URL) Query

```
func (u *URL) Query() Values
```

Query方法解析RawQuery字段并返回其表示的Values类型键值对。

func (*URL) RequestURI

```
func (u *URL) RequestURI() string
```

RequestURI方法返回编码好的path?query或opaque?query字符串，用在HTTP请求里。

func (*URL) String

```
func (u *URL) String() string
```

String将URL重构为一个合法URL字符串。

func (*URL) Parse

```
func (u *URL) Parse(ref string) (*URL, error)
```

Parse方法以u为上下文来解析一个URL，ref可以是绝对或相对URL。

本方法解析失败会返回nil, err；否则返回结果和ResolveReference一致。

func (*URL) ResolveReference

```
func (u *URL) ResolveReference(ref *URL) *URL
```

本方法根据一个绝对URI将一个URI补全为一个绝对URI，参见RFC 3986 节 5.2。参数ref可以是绝对URI或者相对URI。ResolveReference总是返回一个新的URL实例，即使该实例和u或者ref完全一样。如果ref是绝对URI，本方法会忽略参照URI并返回ref的一个拷贝。

type Userinfo

```
type Userinfo struct {
    // 内含隐藏或非导出字段
}
```

Userinfo类型是一个URL的用户名和密码细节的一个不可修改的封装。一个真实存在的Userinfo值必须保证有用户名（但根据 RFC 2396可以是空字符串）以及一个可选的密码。

func User

```
func User(username string) *Userinfo
```

User函数返回一个用户名设置为username的不设置密码的*Userinfo。

func UserPassword

```
func UserPassword(username, password string) *Userinfo
```

UserPassword函数返回一个用户名设置为username、密码设置为password的*Userinfo。

这个函数应该只用于老式的站点，因为风险很大，不建议使用，参见RFC 2396。

func (*Userinfo) Username

```
func (u *Userinfo) Username() string
```

Username方法返回用户名。

func (*Userinfo) Password

```
func (u *Userinfo) Password() (string, bool)
```

如果设置了密码返回密码和真，否则会返回假。

func (*Userinfo) String

```
func (u *Userinfo) String() string
```

String方法返回编码后的用户信息，格式为"username[:password]"。

type Values

```
type Values map[string][]string
```

Values将建映射到值的列表。它一般用于查询的参数和表单的属性。不同于http.Header这个字典类型，Values的键是大小写敏感的。

Example

```
v := url.Values{}
v.Set("name", "Ava")
v.Add("friend", "Jess")
v.Add("friend", "Sarah")
v.Add("friend", "Zoe")
// v.Encode() == "name=Ava&friend=Jess&friend=Sarah&friend=Zoe"
fmt.Println(v.Get("name"))
fmt.Println(v.Get("friend"))
fmt.Println(v["friend"])
```

Output:

```
Ava
Jess
[Jess Sarah Zoe]
```

func ParseQuery

```
func ParseQuery(query string) (m Values, err error)
```

ParseQuery函数解析一个URL编码的查询字符串，并返回可以表示该查询的Values类型的字典。本函数总是返回一个包含了所有合法查询参数的非nil字典，err用来描述解码时遇到的（如果有）第一个错误。

func (Values) Get

```
func (v Values) Get(key string) string
```

Get会获取key对应的值集的第一个值。如果没有对应key的值集会返回空字符串。获取值集请直接用map。

func (Values) Set

```
func (v Values) Set(key, value string)
```

Set方法将key对应的值集设为只有value，它会替换掉已有的值集。

func (Values) Add

```
func (v Values) Add(key, value string)
```

Add将value添加到key关联的值集里原有的值的后面。

func (Values) Del

```
func (v Values) Del(key string)
```

Del删除key关联的值集。

func (Values) Encode

```
func (v Values) Encode() string
```

Encode方法将v编码为url编码格式("bar=baz&foo=quux")，编码时会以键进行排序。

package os

```
import "os"
```

os包提供了操作系统函数的不依赖平台的接口。设计为Unix风格的，虽然错误处理是go风格的；失败的调用会返回错误值而非错误码。通常错误值里包含更多信息。例如，如果某个使用一个文件名的调用（如Open、Stat）失败了，打印错误时会包含该文件名，错误类型将为*PathError，其内部可以解包获得更多信息。

os包的接口规定为在所有操作系统中都是一致的。非公用的属性可以从操作系统特定的syscall包获取。

下面是一个简单的例子，打开一个文件并从中读取一些数据：

```
file, err := os.Open("file.go") // For read access.
if err != nil {
    log.Fatal(err)
}
```

如果打开失败，错误字符串是自解释的，例如：

```
open file.go: no such file or directory
```

文件的信息可以读取进一个[]byte切片。Read和Write方法从切片参数获取其内的字节数。

```
data := make([]byte, 100)
count, err := file.Read(data)
if err != nil {
    log.Fatal(err)
}
fmt.Printf("read %d bytes: %q\n", count, data[:count])
```

Index

Constants

```
const (
    O_RDONLY int = syscall.O_RDONLY // 只读模式打开文件
    O_WRONLY int = syscall.O_WRONLY // 只写模式打开文件
    O_RDWR   int = syscall.O_RDWR   // 读写模式打开文件
    O_APPEND int = syscall.O_APPEND // 写操作时将数据附加到文件尾部
    O_CREATE int = syscall.O_CREAT  // 如果不存在将创建一个新文件
    O_EXCL   int = syscall.O_EXCL   // 和O_CREATE配合使用，文件必须不存
    O_SYNC   int = syscall.O_SYNC   // 打开文件用于同步I/O
    O_TRUNC  int = syscall.O_TRUNC  // 如果可能，打开时清空文件
)
```

用于包装底层系统的参数用于Open函数，不是所有的flag都能在特定系统里使用的。

```
const (
    SEEK_SET int = 0 // 相对于文件起始位置seek
    SEEK_CUR int = 1 // 相对于文件当前位置seek
    SEEK_END int = 2 // 相对于文件结尾位置seek
)
```

指定Seek函数从何处开始搜索（即相对位置）

```
const (
    PathSeparator     = '/' // 操作系统指定的路径分隔符
    PathListSeparator = ':' // 操作系统指定的表分隔符
)
```

```
const DevNull = "/dev/null"
```

DevNull是操作系统空设备的名字。在类似Unix的操作系统中，是"/dev/null"；在Windows中，为"NUL"。

Variables

```
var (
    ErrInvalid    = errors.New("invalid argument")
    ErrPermission = errors.New("permission denied")
    ErrExist      = errors.New("file already exists")
    ErrNotExist   = errors.New("file does not exist")
)
```

一些可移植的、共有的系统调用错误。

```
var (
    Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

Stdin、Stdout和Stderr是指向标准输入、标准输出、标准错误输出的文件描述符。

```
var Args []string
```

Args保管了命令行参数，第一个是程序名。

func Hostname

```
func Hostname() (name string, err error)
```

Hostname返回内核提供的主机名。

func Getpagesize

```
func Getpagesize() int
```

Getpagesize返回底层的系统内存页的尺寸。

func Environ

```
func Environ() []string
```

Environ返回表示环境变量的格式为"key=value"的字符串的切片拷贝。

func Getenv

```
func Getenv(key string) string
```

Getenv检索并返回名为key的环境变量的值。如果不存在该环境变量会返回空字符串。

func Setenv

```
func Setenv(key, value string) error
```

Setenv设置名为key的环境变量。如果出错会返回该错误。

func Clearenv

```
func Clearenv()
```

Clearenv删除所有环境变量。

func Exit

```
func Exit(code int)
```

Exit让当前程序以给出的状态码code退出。一般来说，状态码0表示成功，非0表示出错。程序会立刻终止，defer的函数不会被执行。

func Expand

```
func Expand(s string, mapping func(string) string) string
```

Expand函数替换s中的${var}或$var为mapping(var)。例如，os.ExpandEnv(s)等价于os.Expand(s, os.Getenv)。

func ExpandEnv

```
func ExpandEnv(s string) string
```

ExpandEnv函数替换s中的${var}或$var为名为var 的环境变量的值。引用未定义环境变量会被替换为空字符串。

func Getuid

```
func Getuid() int
```

Getuid返回调用者的用户ID。

func Geteuid

```
func Geteuid() int
```

Geteuid返回调用者的有效用户ID。

func Getgid

```
func Getgid() int
```

Getgid返回调用者的组ID。

func Getegid

```
func Getegid() int
```

Getegid返回调用者的有效组ID。

func Getgroups

```
func Getgroups() ([]int, error)
```

Getgroups返回调用者所属的所有用户组的组ID。

func Getpid

```
func Getpid() int
```

Getpid返回调用者所在进程的进程ID。

func Getppid

```
func Getppid() int
```

Getppid返回调用者所在进程的父进程的进程ID。

type Signal

```
type Signal interface {
    String() string
    Signal() // 用来区分其他实现了Stringer接口的类型
}
```

Signal代表一个操作系统信号。一般其底层实现是依赖于操作系统的：在Unix中，它是syscall.Signal类型。

```
var (
    Interrupt Signal = syscall.SIGINT
    Kill      Signal = syscall.SIGKILL
)
```

仅有的肯定会被所有操作系统提供的信号，Interrupt（中断信号）和Kill（强制退出信号）。

type PathError

```
type PathError struct {
    Op   string
    Path string
    Err  error
}
```

PathError记录一个错误，以及导致错误的路径。

func (*PathError) Error

```
func (e *PathError) Error() string
```

type LinkError

```
type LinkError struct {
    Op  string
    Old string
    New string
    Err error
}
```

LinkError记录在Link、Symlink、Rename系统调用时出现的错误，以及导致错误的路径。

func (*LinkError) Error

```
func (e *LinkError) Error() string
```

type SyscallError

```
type SyscallError struct {
    Syscall string
    Err     error
}
```

SyscallError记录某个系统调用出现的错误。

func (*SyscallError) Error

```
func (e *SyscallError) Error() string
```

func NewSyscallError

```
func NewSyscallError(syscall string, err error) error
```

NewSyscallError返回一个指定系统调用名称和错误细节的SyscallError。如果err为nil，本函数会返回nil。

type FileMode

```
type FileMode uint32
```

FileMode代表文件的模式和权限位。这些字位在所有的操作系统都有相同的含义，因此文件的信息可以在不同的操作系统之间安全的移植。不是所有的位都能用于所有的系统，唯一共有的是用于表示目录的ModeDir位。

```
const (
    // 单字符是被String方法用于格式化的属性缩写。
    ModeDir        FileMode = 1 << (32 - 1 - iota) // d: 目录
    ModeAppend                                     // a: 只能写入，且
    ModeExclusive                                  // l: 用于执行
    ModeTemporary                                  // T: 临时文件（非
    ModeSymlink                                    // L: 符号链接（不
    ModeDevice                                     // D: 设备
    ModeNamedPipe                                  // p: 命名管道（F
    ModeSocket                                     // S: Unix域sock
    ModeSetuid                                     // u: 表示文件具有
    ModeSetgid                                     // g: 表示文件具有
    ModeCharDevice                                 // c: 字符设备，需
    ModeSticky                                     // t: 只有root/创
    // 覆盖所有类型位（用于通过&获取类型位），对普通文件，所有这些位都不应被设
    ModeType = ModeDir | ModeSymlink | ModeNamedPipe | ModeSocket
    ModePerm FileMode = 0777 // 覆盖所有Unix权限位（用于通过&获取类型位）
)
```

这些被定义的位是FileMode最重要的位。另外9个不重要的位为标准Unix rwxrwxrwx权限（任何人都可读、写、运行）。这些（重要）位的值应被视为公共API的一部分，可能会用于线路协议或硬盘标识：它们不能被修改，但可以添加新的位。

func (FileMode) IsDir

```
func (m FileMode) IsDir() bool
```

IsDir报告m是否是一个目录。

func (FileMode) IsRegular

```
func (m FileMode) IsRegular() bool
```

IsRegular报告m是否是一个普通文件。

func (FileMode) Perm

```
func (m FileMode) Perm() FileMode
```

Perm方法返回m的Unix权限位。

func (FileMode) String

```
func (m FileMode) String() string
```

type FileInfo

```
type FileInfo interface {
    Name() string       // 文件的名字（不含扩展名）
    Size() int64        // 普通文件返回值表示其大小；其他文件的返回值含义各
    Mode() FileMode     // 文件的模式位
    ModTime() time.Time // 文件的修改时间
    IsDir() bool        // 等价于Mode().IsDir()
    Sys() interface{}   // 底层数据来源（可以返回nil）
}
```

FileInfo用来描述一个文件对象。

func Stat

```
func Stat(name string) (fi FileInfo, err error)
```

Stat返回一个描述name指定的文件对象的FileInfo。如果指定的文件对象是一个符号链接，返回的FileInfo描述该符号链接指向的文件的信息，本函数会尝试跳转该链接。如果出错，返回的错误值为*PathError类型。

func Lstat

```
func Lstat(name string) (fi FileInfo, err error)
```

Lstat返回一个描述name指定的文件对象的FileInfo。如果指定的文件对象是一个符号链接，返回的FileInfo描述该符号链接的信息，本函数不会试图跳转该链接。如果出错，返回的错误值为*PathError类型。

func IsPathSeparator

```
func IsPathSeparator(c uint8) bool
```

IsPathSeparator返回字符c是否是一个路径分隔符。

func IsExist

```
func IsExist(err error) bool
```

返回一个布尔值说明该错误是否表示一个文件或目录已经存在。ErrExist和一些系统调用错误会使它返回真。

func IsNotExist

```
func IsNotExist(err error) bool
```

返回一个布尔值说明该错误是否表示一个文件或目录不存在。ErrNotExist和一些系统调用错误会使它返回真。

func IsPermission

```
func IsPermission(err error) bool
```

返回一个布尔值说明该错误是否表示因权限不足要求被拒绝。ErrPermission和一些系统调用错误会使它返回真。

func Getwd

```
func Getwd() (dir string, err error)
```

Getwd返回一个对应当前工作目录的根路径。如果当前目录可以经过多条路径抵达（因为硬链接），Getwd会返回其中一个。

func Chdir

```
func Chdir(dir string) error
```

Chdir将当前工作目录修改为dir指定的目录。如果出错，会返回*PathError底层类型的错误。

func Chmod

```
func Chmod(name string, mode FileMode) error
```

Chmod修改name指定的文件对象的mode。如果name指定的文件是一个符号链接，它会修改该链接的目的地文件的mode。如果出错，会返回*PathError底层类型的错误。

func Chown

```
func Chown(name string, uid, gid int) error
```

Chmod修改name指定的文件对象的用户id和组id。如果name指定的文件是一个符号链接，它会修改该链接的目的地文件的用户id和组id。如果出错，会返回*PathError底层类型的错误。

func Lchown

```
func Lchown(name string, uid, gid int) error
```

Chmod修改name指定的文件对象的用户id和组id。如果name指定的文件是一个符号链接，它会修改该符号链接自身的用户id和组id。如果出错，会返回*PathError底层类型的错误。

func Chtimes

```
func Chtimes(name string, atime time.Time, mtime time.Time) error
```

Chtimes修改name指定的文件对象的访问时间和修改时间，类似Unix的utime()或utimes()函数。底层的文件系统可能会截断/舍入时间单位到更低的精确度。如果出错，会返回*PathError底层类型的错误。

func Mkdir

```
func Mkdir(name string, perm FileMode) error
```

Mkdir使用指定的权限和名称创建一个目录。如果出错，会返回*PathError底层类型的错误。

func MkdirAll

```
func MkdirAll(path string, perm FileMode) error
```

MkdirAll使用指定的权限和名称创建一个目录，包括任何必要的上级目录，并返回nil，否则返回错误。权限位perm会应用在每一个被本函数创建的目录上。如果path指定了一个已经存在的目录，MkdirAll不做任何操作并返回nil。

func Rename

```
func Rename(oldpath, newpath string) error
```

Rename修改一个文件的名字，移动一个文件。可能会有一些个操作系统特定的限制。

func Truncate

```
func Truncate(name string, size int64) error
```

Truncate修改name指定的文件的大小。如果该文件为一个符号链接，将修改链接指向的文件的大小。如果出错，会返回*PathError底层类型的错误。

func Remove

```
func Remove(name string) error
```

Remove删除name指定的文件或目录。如果出错，会返回*PathError底层类型的错误。

func RemoveAll

```
func RemoveAll(path string) error
```

RemoveAll删除path指定的文件，或目录及它包含的任何下级对象。它会尝试删除所有东西，除非遇到错误并返回。如果path指定的对象不存在，RemoveAll会返回nil而不返回错误。

func Readlink

```
func Readlink(name string) (string, error)
```

Readlink获取name指定的符号链接文件指向的文件的路径。如果出错，会返回*PathError底层类型的错误。

func Symlink

```
func Symlink(oldname, newname string) error
```

Symlink创建一个名为newname指向oldname的符号链接。如果出错，会返回* LinkError底层类型的错误。

func Link

```
func Link(oldname, newname string) error
```

Link创建一个名为newname指向oldname的硬链接。如果出错，会返回* LinkError底层类型的错误。

func SameFile

```
func SameFile(fi1, fi2 FileInfo) bool
```

SameFile返回fi1和fi2是否在描述同一个文件。例如，在Unix这表示二者底层结构的设备和索引节点是相同的；在其他系统中可能是根据路径名确定的。SameFile应只使用本包Stat函数返回的FileInfo类型值为参数，其他情况下，它会返回假。

func TempDir

```
func TempDir() string
```

TempDir返回一个用于保管临时文件的默认目录。

type File

```
type File struct {
    // 内含隐藏或非导出字段
}
```

File代表一个打开的文件对象。

func Create

```
func Create(name string) (file *File, err error)
```

Create采用模式0666（任何人都可读写，不可执行）创建一个名为name的文件，如果文件已存在会截断它（为空文件）。如果成功，返回的文件对象可用于I/O；对应的文件描述符具有O_RDWR模式。如果出错，错误底层类型是*PathError。

func Open

```
func Open(name string) (file *File, err error)
```

Open打开一个文件用于读取。如果操作成功，返回的文件对象的方法可用于读取数据；对应的文件描述符具有O_RDONLY模式。如果出错，错误底层类型是*PathError。

func OpenFile

```
func OpenFile(name string, flag int, perm FileMode) (file *File, er
```

OpenFile是一个更一般性的文件打开函数，大多数调用者都应用Open或Create代替本函数。它会使用指定的选项（如O_RDONLY等）、指定的模式（如0666等）打开指定名称的文件。如果操作成功，返回的文件对象可用于I/O。如果出错，错误

底层类型是*PathError。

func NewFile

```
func NewFile(fd uintptr, name string) *File
```

NewFile使用给出的Unix文件描述符和名称创建一个文件。

func Pipe

```
func Pipe() (r *File, w *File, err error)
```

Pipe返回一对关联的文件对象。从r的读取将返回写入w的数据。本函数会返回两个文件对象和可能的错误。

func (*File) Name

```
func (f *File) Name() string
```

Name方法返回（提供给Open/Create等方法的）文件名称。

func (*File) Stat

```
func (f *File) Stat() (fi FileInfo, err error)
```

Stat返回描述文件f的FileInfo类型值。如果出错，错误底层类型是*PathError。

func (*File) Fd

```
func (f *File) Fd() uintptr
```

Fd返回与文件f对应的整数类型的Unix文件描述符。

func (*File) Chdir

```
func (f *File) Chdir() error
```

Chdir将当前工作目录修改为f，f必须是一个目录。如果出错，错误底层类型是*PathError。

func (*File) Chmod

```
func (f *File) Chmod(mode FileMode) error
```

Chmod修改文件的模式。如果出错，错误底层类型是*PathError。

func (*File) Chown

```
func (f *File) Chown(uid, gid int) error
```

Chown修改文件的用户ID和组ID。如果出错，错误底层类型是*PathError。

func (*File) Readdir

```
func (f *File) Readdir(n int) (fi []FileInfo, err error)
```

Readdir读取目录f的内容，返回一个有n个成员的[]FileInfo，这些FileInfo是被Lstat返回的，采用目录顺序。对本函数的下一次调用会返回上一次调用剩余未读取的内容的信息。

如果n>0，Readdir函数会返回一个最多n个成员的切片。这时，如果Readdir返回一个空切片，它会返回一个非nil的错误说明原因。如果到达了目录f的结尾，返回值err会是io.EOF。

如果n<=0，Readdir函数返回目录中剩余所有文件对象的FileInfo构成的切片。此时，如果Readdir调用成功（读取所有内容直到结尾），它会返回该切片和nil的错误值。如果在到达结尾前遇到错误，会返回之前成功读取的FileInfo构成的切片和该错误。

func (*File) Readdirnames

```
func (f *File) Readdirnames(n int) (names []string, err error)
```

Readdir读取目录f的内容，返回一个有n个成员的[]string，切片成员为目录中文件对象的名字，采用目录顺序。对本函数的下一次调用会返回上一次调用剩余未读取的内容的信息。

如果n>0，Readdir函数会返回一个最多n个成员的切片。这时，如果Readdir返回一个空切片，它会返回一个非nil的错误说明原因。如果到达了目录f的结尾，返回值err会是io.EOF。

如果n<=0，Readdir函数返回目录中剩余所有文件对象的名字构成的切片。此时，如果Readdir调用成功（读取所有内容直到结尾），它会返回该切片和nil的错误值。如果在到达结尾前遇到错误，会返回之前成功读取的名字构成的切片和该错误。

func (*File) Truncate

```
func (f *File) Truncate(size int64) error
```

Truncate改变文件的大小，它不会改变I/O的当前位置。 如果截断文件，多出的部分就会被丢弃。如果出错，错误底层类型是*PathError。

func (*File) Read

```
func (f *File) Read(b []byte) (n int, err error)
```

Read方法从f中读取最多len(b)字节数据并写入b。它返回读取的字节数和可能遇到的任何错误。文件终止标志是读取0个字节且返回值err为io.EOF。

func (*File) ReadAt

```
func (f *File) ReadAt(b []byte, off int64) (n int, err error)
```

ReadAt从指定的位置（相对于文件开始位置）读取len(b)字节数据并写入b。它返回读取的字节数和可能遇到的任何错误。当n<len(b)时，本方法总是会返回错误；如果是因为到达文件结尾，返回值err会是io.EOF。

func (*File) Write

```
func (f *File) Write(b []byte) (n int, err error)
```

Write向文件中写入len(b)字节数据。它返回写入的字节数和可能遇到的任何错误。如果返回值n!=len(b)，本方法会返回一个非nil的错误。

func (*File) WriteString

```
func (f *File) WriteString(s string) (ret int, err error)
```

WriteString类似Write，但接受一个字符串参数。

func (*File) WriteAt

```
func (f *File) WriteAt(b []byte, off int64) (n int, err error)
```

WriteAt在指定的位置（相对于文件开始位置）写入len(b)字节数据。它返回写入的字节数和可能遇到的任何错误。如果返回值n!=len(b)，本方法会返回一个非nil的错误。

func (*File) Seek

```
func (f *File) Seek(offset int64, whence int) (ret int64, err erro
```

Seek设置下一次读/写的位置。offset为相对偏移量，而whence决定相对位置：0为相对文件开头，1为相对当前位置，2为相对文件结尾。它返回新的偏移量（相对开头）和可能的错误。

func (*File) Sync

```
func (f *File) Sync() (err error)
```

Sync递交文件的当前内容进行稳定的存储。一般来说，这表示将文件系统的最近写入的数据在内存中的拷贝刷新到硬盘中稳定保存。

func (*File) Close

```
func (f *File) Close() error
```

Close关闭文件f，使文件不能用于读写。它返回可能出现的错误。

type ProcAttr

```
type ProcAttr struct {
    // 如果Dir非空，子进程会在创建进程前先进入该目录。（即设为当前工作目录）
    Dir string
    // 如果Env非空，它会作为新进程的环境变量。必须采用Environ返回值的格式。
    // 如果Env为空字符串，将使用Environ函数的返回值。
    Env []string
    // Files指定被新进程继承的活动文件对象。
    // 前三个绑定为标准输入、标准输出、标准错误输出。
    // 依赖底层操作系统的实现可能会支持额外的数据出入途径。
    // nil条目相当于在进程开始时关闭的文件对象。
    Files []*File
    // 操作系统特定的创建属性。
    // 注意设置本字段意味着你的程序可能会运作失常甚至在某些操作系统中无法通过编
    Sys *syscall.SysProcAttr
}
```

ProcAttr保管将被StartProcess函数用于一个新进程的属性。

type Process

```
type Process struct {
    Pid int
    // 内含隐藏或非导出字段
}
```

Process保管一个被StarProcess创建的进程的信息。

func FindProcess

```
func FindProcess(pid int) (p *Process, err error)
```

FindProcess根据进程id查找一个运行中的进程。函数返回的进程对象可以用于获取其关于底层操作系统进程的信息。

func StartProcess

```
func StartProcess(name string, argv []string, attr *ProcAttr) (*Pro
```

StartProcess使用提供的属性、程序名、命令行参数开始一个新进程。StartProcess函数是一个低水平的接口。os/exec包提供了高水平的接口，应该尽量使用该包。如果出错，错误的底层类型会是*PathError。

func (*Process) Signal

```
func (p *Process) Signal(sig Signal) error
```

Signal方法向进程发送一个信号。在windows中向进程发送Interrupt信号尚未实现。

func (*Process) Kill

```
func (p *Process) Kill() error
```

Kill让进程立刻退出。

func (*Process) Wait

```
func (p *Process) Wait() (*ProcessState, error)
```

Wait方法阻塞直到进程退出，然后返回一个描述ProcessState描述进程的状态和可能的错误。Wait方法会释放绑定到进程p的所有资源。在大多数操作系统中，进程p必须是当前进程的子进程，否则会返回错误。

func (*Process) Release

```
func (p *Process) Release() error
```

Release释放进程p绑定的所有资源， 使它们（资源）不能再被（进程p）使用。只有没有调用Wait方法时才需要调用本方法。

type ProcessState

```
type ProcessState struct {
    // 内含隐藏或非导出字段
}
```

ProcessState保管Wait函数报告的某个已退出进程的信息。

func (*ProcessState) Pid

```
func (p *ProcessState) Pid() int
```

Pi返回一个已退出的进程的进程id。

func (*ProcessState) Exited

```
func (p *ProcessState) Exited() bool
```

Exited报告进程是否已退出。

func (*ProcessState) Success

```
func (p *ProcessState) Success() bool
```

Success报告进程是否成功退出，如在Unix里以状态码0退出。

func (*ProcessState) SystemTime

```
func (p *ProcessState) SystemTime() time.Duration
```

SystemTime返回已退出进程及其子进程耗费的系统CPU时间。

func (*ProcessState) UserTime

```
func (p *ProcessState) UserTime() time.Duration
```

UserTime返回已退出进程及其子进程耗费的用户CPU时间。

func (*ProcessState) Sys

```
func (p *ProcessState) Sys() interface{}
```

Sys返回该已退出进程系统特定的退出信息。需要将其类型转换为适当的底层类型，如Unix里转换为*syscall.WaitStatus类型以获取其内容。

func (*ProcessState) SysUsage

```
func (p *ProcessState) SysUsage() interface{}
```

SysUsage返回该已退出进程系统特定的资源使用信息。需要将其类型转换为适当的底层类型，如Unix里转换为*syscall.Rusage类型以获取其内容。

func (*ProcessState) String

```
func (p *ProcessState) String() string
```

package exec

```
import "os/exec"
```

exec包执行外部命令。它包装了os.StartProcess函数以便更容易的修正输入和输出，使用管道连接I/O，以及作其它的一些调整。

Index

Examples

Variables

```
var ErrNotFound = errors.New("executable file not found in $PATH")
```

如果路径搜索没有找到可执行文件时，就会返回本错误。

type Error

```
type Error struct {
    Name string
    Err  error
}
```

Error类型记录执行失败的程序名和失败的原因。

func (*Error) Error

```
func (e *Error) Error() string
```

type ExitError

```
type ExitError struct {
    *os.ProcessState
}
```

ExitError报告某个命令的一次未成功的返回。

func (*ExitError) Error

```
func (e *ExitError) Error() string
```

func LookPath

```
func LookPath(file string) (string, error)
```

在环境变量PATH指定的目录中搜索可执行文件，如file中有斜杠，则只在当前目录搜索。返回完整路径或者相对于当前目录的一个相对路径。

Example

```
path, err := exec.LookPath("fortune")
if err != nil {
    log.Fatal("installing fortune is in your future")
}
fmt.Printf("fortune is available at %s\n", path)
```

type Cmd

```
type Cmd struct {
    // Path是将要执行的命令的路径。
    //
    // 该字段不能为空，如为相对路径会相对于Dir字段。
    Path string
    // Args保管命令的参数，包括命令名作为第一个参数；如果为空切片或者nil，相兰
    //
    // 典型用法下，Path和Args都应被Command函数设定。
    Args []string
    // Env指定进程的环境，如为nil，则是在当前进程的环境下执行。
    Env []string
    // Dir指定命令的工作目录。如为空字符串，会在调用者的进程当前目录下执行。
    Dir string
    // Stdin指定进程的标准输入，如为nil，进程会从空设备读取（os.DevNull）
    Stdin io.Reader
    // Stdout和Stderr指定进程的标准输出和标准错误输出。
    //
    // 如果任一个为nil，Run方法会将对应的文件描述符关联到空设备（os.DevNull
    //
    // 如果两个字段相同，同一时间最多有一个线程可以写入。
    Stdout io.Writer
    Stderr io.Writer
    // ExtraFiles指定额外被新进程继承的已打开文件流，不包括标准输入、标准输出
    // 如果本字段非nil，entry i会变成文件描述符3+i。
    //
    // BUG: 在OS X 10.6系统中，子进程可能会继承不期望的文件描述符。
    // http://golang.org/issue/2603
    ExtraFiles []*os.File
    // SysProcAttr保管可选的、各操作系统特定的sys执行属性。
    // Run方法会将它作为os.ProcAttr的Sys字段传递给os.StartProcess函数。
    SysProcAttr *syscall.SysProcAttr
    // Process是底层的，只执行一次的进程。
    Process *os.Process
    // ProcessState包含一个已经存在的进程的信息，只有在调用Wait或Run后才可
    ProcessState *os.ProcessState
    // 内含隐藏或非导出字段
}
```

Cmd代表一个正在准备或者在执行中的外部命令。

func Command

```
func Command(name string, arg ...string) *Cmd
```

函数返回一个*Cmd，用于使用给出的参数执行name指定的程序。返回值只设定了Path和Args两个参数。

如果name不含路径分隔符，将使用LookPath获取完整路径；否则直接使用name。参数arg不应包含命令名。

Example

```
cmd := exec.Command("tr", "a-z", "A-Z")
cmd.Stdin = strings.NewReader("some input")
var out bytes.Buffer
cmd.Stdout = &out
err := cmd.Run()
if err != nil {
    log.Fatal(err)
}
fmt.Printf("in all caps: %q\n", out.String())
```

func (*Cmd) StdinPipe

```
func (c *Cmd) StdinPipe() (io.WriteCloser, error)
```

StdinPipe方法返回一个在命令Start后与命令标准输入关联的管道。Wait方法获知命令结束后会关闭这个管道。必要时调用者可以调用Close方法来强行关闭管道，例如命令在输入关闭后才会执行返回时需要显式关闭管道。

func (*Cmd) StdoutPipe

```
func (c *Cmd) StdoutPipe() (io.ReadCloser, error)
```

StdoutPipe方法返回一个在命令Start后与命令标准输出关联的管道。Wait方法获知命令结束后会关闭这个管道，一般不需要显式的关闭该管道。但是在从管道读取完全部数据之前调用Wait是错误的；同样使用StdoutPipe方法时调用Run函数也是错误的。参见下例：

Example

```
cmd := exec.Command("echo", "-n", `{"Name": "Bob", "Age": 32}`)
stdout, err := cmd.StdoutPipe()
if err != nil {
    log.Fatal(err)
}
if err := cmd.Start(); err != nil {
    log.Fatal(err)
}
var person struct {
    Name string
    Age  int
}
if err := json.NewDecoder(stdout).Decode(&person); err != nil {
    log.Fatal(err)
}
if err := cmd.Wait(); err != nil {
    log.Fatal(err)
}
fmt.Printf("%s is %d years old\n", person.Name, person.Age)
```

func (*Cmd) StderrPipe

```
func (c *Cmd) StderrPipe() (io.ReadCloser, error)
```

StderrPipe方法返回一个在命令Start后与命令标准错误输出关联的管道。Wait方法获知命令结束后会关闭这个管道，一般不需要显式的关闭该管道。但是在从管道读取完全部数据之前调用Wait是错误的；同样使用StderrPipe方法时调用Run函数也是错误的。请参照StdoutPipe的例子。

func (*Cmd) Run

```
func (c *Cmd) Run() error
```

Run执行c包含的命令，并阻塞直到完成。

如果命令成功执行，stdin、stdout、stderr的转交没有问题，并且返回状态码为0，方法的返回值为nil；如果命令没有执行或者执行失败，会返回*ExitError类型的错误；否则返回的error可能是表示I/O问题。

func (*Cmd) Start

```
func (c *Cmd) Start() error
```

Start开始执行c包含的命令，但并不会等待该命令完成即返回。Wait方法会返回命令的返回状态码并在命令返回后释放相关的资源。

Example

```
cmd := exec.Command("sleep", "5")
err := cmd.Start()
if err != nil {
    log.Fatal(err)
}
log.Printf("Waiting for command to finish...")
err = cmd.Wait()
log.Printf("Command finished with error: %v", err)
```

func (*Cmd) Wait

```
func (c *Cmd) Wait() error
```

Wait会阻塞直到该命令执行完成，该命令必须是被Start方法开始执行的。

如果命令成功执行，stdin、stdout、stderr的转交没有问题，并且返回状态码为0，方法的返回值为nil；如果命令没有执行或者执行失败，会返回*ExitError类型的错误；否则返回的error可能是表示I/O问题。Wait方法会在命令返回后释放相关的资源。

func (*Cmd) Output

```
func (c *Cmd) Output() ([]byte, error)
```

执行命令并返回标准输出的切片。

Example

```
out, err := exec.Command("date").Output()
if err != nil {
    log.Fatal(err)
}
fmt.Printf("The date is %s\n", out)
```

func (*Cmd) CombinedOutput

```
func (c *Cmd) CombinedOutput() ([]byte, error)
```

执行命令并返回标准输出和错误输出合并的切片。

package signal

`import "os/signal"`

signal包实现了对输入信号的访问。

Index

Examples

func Notify

```
func Notify(c chan<- os.Signal, sig ...os.Signal)
```

Notify函数让signal包将输入信号转发到c。如果没有列出要传递的信号，会将所有输入信号传递到c；否则只传递列出的输入信号。

signal包不会为了向c发送信息而阻塞（就是说如果发送时c阻塞了，signal包会直接放弃）：调用者应该保证c有足够的缓存空间可以跟上期望的信号频率。对使用单一信号用于通知的通道，缓存为1就足够了。

可以使用同一通道多次调用Notify：每一次都会扩展该通道接收的信号集。唯一从信号集去除信号的方法是调用Stop。可以使用同一信号和不同通道多次调用Notify：每一个通道都会独立接收到该信号的一个拷贝。

Example

```
// Set up channel on which to send signal notifications.
// We must use a buffered channel or risk missing the signal
// if we're not ready to receive when the signal is sent.
c := make(chan os.Signal, 1)
signal.Notify(c, os.Interrupt, os.Kill)
// Block until a signal is received.
s := <-c
fmt.Println("Got signal:", s)
```

func Stop

```
func Stop(c chan<- os.Signal)
```

Stop函数让signal包停止向c转发信号。它会取消之前使用c调用的所有Notify的效果。当Stop返回后，会保证c不再接收到任何信号。

Bugs

☞ 本包还没在Plan 9上实现。

package user

```
import "os/user"
```

user包允许通过名称或ID查询用户帐户。

Index

type UnknownUserError

```
type UnknownUserError string
```

当找不到用户时，Lookup会返回UnknownUserError

func (UnknownUserError) Error

```
func (e UnknownUserError) Error() string
```

type UnknownUserIdError

```
type UnknownUserIdError int
```

当找不到用户时，LookupId会返回UnknownUserIdError

func (UnknownUserIdError) Error

```
func (e UnknownUserIdError) Error() string
```

type User

```
type User struct {
    Uid      string // 用户ID
    Gid      string // 初级组ID
    Username string
    Name     string
    HomeDir  string
}
```

User代表一个用户帐户。

在posix系统中Uid和Gid字段分别包含代表uid和gid的十进制数字。在windows系统中Uid和Gid包含字符串格式的安全标识符（SID）。在Plan 9系统中，Uid、Gid、Username和Name字段是/dev/user的内容。

func Current

```
func Current() (*User, error)
```

返回当前的用户帐户。

func Lookup

```
func Lookup(username string) (*User, error)
```

根据用户名查询用户。

func LookupId

```
func LookupId(uid string) (*User, error)
```

根据用户ID查询用户。

package path

```
import "path"
```

path实现了对斜杠分隔的路径的实用操作函数。

Index

Examples

Variables

```
var ErrBadPattern = errors.New("syntax error in pattern")
```

ErrBadPattern表示一个glob模式匹配字符串的格式错误。

func IsAbs

```
func IsAbs(path string) bool
```

IsAbs返回路径是否是一个绝对路径。

Example

```
fmt.Println(path.IsAbs("/dev/null"))
```

Output:

```
true
```

func Split

```
func Split(path string) (dir, file string)
```

Split函数将路径从最后一个斜杠后面位置分隔为两个部分（dir和file）并返回。如果路径中没有斜杠，函数返回值dir会设为空字符串，file会设为path。两个返回值满足path == dir+file。

Example

```
fmt.Println(path.Split("static/myfile.css"))
```

Output:

```
static/ myfile.css
```

func Join

```
func Join(elem ...string) string
```

Join函数可以将任意数量的路径元素放入一个单一路径里，会根据需要添加斜杠。结果是经过简化的，所有的空字符串元素会被忽略。

Example

```
fmt.Println(path.Join("a", "b", "c"))
```

Output:

```
a/b/c
```

func Dir

```
func Dir(path string) string
```

Dir返回路径除去最后一个路径元素的部分，即该路径最后一个元素所在的目录。在使用Split去掉最后一个元素后，会简化路径并去掉末尾的斜杠。如果路径是空字符串，会返回"."；如果路径由1到多个斜杠后跟0到多个非斜杠字符组成，会返回"/"；其他任何情况下都不会返回以斜杠结尾的路径。

Example

```
fmt.Println(path.Dir("/a/b/c"))
```

Output:

```
/a/b
```

func Base

```
func Base(path string) string
```

Base函数返回路径的最后一个元素。在提取元素前会求掉末尾的斜杠。如果路径是""，会返回"."；如果路径是只有一个斜杆构成，会返回"/"。

Example

```
fmt.Println(path.Base("/a/b"))
```

Output:

```
b
```

func Ext

```
func Ext(path string) string
```

Ext函数返回path文件扩展名。返回值是路径最后一个斜杠分隔出的路径元素的最后一个'.'起始的后缀（包括'.'）。如果该元素没有'.'会返回空字符串。

Example

```
fmt.Println(path.Ext("/a/b/c/bar.css"))
```

Output:

```
.css
```

func Clean

```
func Clean(path string) string
```

Clean函数通过单纯的词法操作返回和path代表同一地址的最短路径。

它会不断的依次应用如下的规则，直到不能再进行任何处理：

```
1\. 将连续的多个斜杠替换为单个斜杠
2\. 剔除每一个.路径名元素（代表当前目录）
3\. 剔除每一个路径内的..路径名元素（代表父目录）和它前面的非..路径名元素
4\. 剔除开始一个根路径的..路径名元素，即将路径开始处的"/.."替换为"/"
```

只有路径代表根地址"/"时才会以斜杠结尾。如果处理的结果是空字符串，Clean会返回"."。

参见http://plan9.bell-labs.com/sys/doc/lexnames.html

Example

```
paths := []string{
    "a/c",
    "a//c",
    "a/c/.",
    "a/c/b/..",
    "/../a/c",
    "/../a/b/../././/c",
}
for _, p := range paths {
    fmt.Printf("Clean(%q) = %q\n", p, path.Clean(p))
}
```

Output:

```
Clean("a/c") = "a/c"
Clean("a//c") = "a/c"
Clean("a/c/.") = "a/c"
Clean("a/c/b/..") = "a/c"
Clean("/../a/c") = "/a/c"
Clean("/../a/b/../././/c") = "/a/c"
```

func Match

```
func Match(pattern, name string) (matched bool, err error)
```

如果name匹配shell文件名模式匹配字符串，Match函数返回真。该模式匹配字符串语法为：

```
pattern:
    { term }
term:
    '*'                                     匹配0或多个非/的字符
    '?'                                     匹配1个非/的字符
    '[' [ '^' ] { character-range } ']'     字符组（必须非空）
    c                                       匹配字符c（c != '*', '?', '
    '\\' c                                  匹配字符c
character-range:
    c           匹配字符c（c != '\\', '-', ']'）
    '\\' c      匹配字符c
    lo '-' hi   匹配区间[lo, hi]内的字符
```

Match要求匹配整个name字符串，而不是它的一部分。只有pattern语法错误时，会返回ErrBadPattern。

package filepath

```
import "path/filepath"
```

filepath包实现了兼容各操作系统的文件路径的实用操作函数。

Index

Examples

Constants

```
const (
    Separator     = os.PathSeparator
    ListSeparator = os.PathListSeparator
)
```

Variables

```
var ErrBadPattern = errors.New("syntax error in pattern")
```

ErrBadPattern表示一个glob模式匹配字符串的格式错误。

```
var SkipDir = errors.New("skip this directory")
```

用作WalkFunc类型的返回值，表示该次调用的path参数指定的目录应被跳过。本错误不应被任何其他函数返回。

func IsAbs

```
func IsAbs(path string) bool
```

IsAbs返回路径是否是一个绝对路径。

func Abs

```
func Abs(path string) (string, error)
```

Abs函数返回path代表的绝对路径，如果path不是绝对路径，会加入当前工作目录以使之成为绝对路径。因为硬链接的存在，不能保证返回的绝对路径是唯一指向该地址的绝对路径。

func Rel

```
func Rel(basepath, targpath string) (string, error)
```

Rel函数返回一个相对路径，将basepath和该路径用路径分隔符连起来的新路径在词法上等价于targpath。也就是说，Join(basepath, Rel(basepath, targpath))等价于targpath本身。如果成功执行，返回值总是相对于basepath的，即使basepath和targpath没有共享的路径元素。如果两个参数一个是相对路径而另一个是绝对路径，或者targpath无法表示为相对于basepath的路径，将返回错误。

Example

```
paths := []string{
    "/a/b/c",
    "/b/c",
    "./b/c",
}
base := "/a"
fmt.Println("On Unix:")
for _, p := range paths {
    rel, err := filepath.Rel(base, p)
    fmt.Printf("%q: %q %v\n", p, rel, err)
}
```

Output:

```
On Unix:
"/a/b/c": "b/c" <nil>
"/b/c": "../b/c" <nil>
"./b/c": "" Rel: can't make b/c relative to /a
```

func SplitList

```
func SplitList(path string) []string
```

将PATH或GOPATH等环境变量里的多个路径分割开（这些路径被OS特定的表分隔符连接起来）。与strings.Split函数的不同之处是：对""，SplitList返回[]string{}，而strings.Split返回[]string{""}。

Example

```
fmt.Println("On Unix:", filepath.SplitList("/a/b/c:/usr/bin"))
```

Output:

```
On Unix: [/a/b/c /usr/bin]
```

func Split

```
func Split(path string) (dir, file string)
```

Split函数将路径从最后一个路径分隔符后面位置分隔为两个部分（dir和file）并返回。如果路径中没有路径分隔符，函数返回值dir会设为空字符串，file会设为path。两个返回值满足path == dir+file。

func Join

```
func Join(elem ...string) string
```

Join函数可以将任意数量的路径元素放入一个单一路径里，会根据需要添加路径分隔符。结果是经过简化的，所有的空字符串元素会被忽略。

func FromSlash

```
func FromSlash(path string) string
```

FromSlash函数将path中的斜杠（'/'）替换为路径分隔符并返回替换结果，多个斜杠会替换为多个路径分隔符。

func ToSlash

```
func ToSlash(path string) string
```

ToSlash函数将path中的路径分隔符替换为斜杠（'/'）并返回替换结果，多个路径分隔符会替换为多个斜杠。

func VolumeName

```
func VolumeName(path string) (v string)
```

VolumeName函数返回最前面的卷名。如Windows系统里提供参数"C:\foo\bar"会返回"C:"；Unix/linux系统的"\host\share\foo"会返回"\host\share"；其他平台会返回""。

func Dir

```
func Dir(path string) string
```

Dir返回路径除去最后一个路径元素的部分，即该路径最后一个元素所在的目录。在使用Split去掉最后一个元素后，会简化路径并去掉末尾的斜杠。如果路径是空字符串，会返回"."；如果路径由1到多个路径分隔符后跟0到多个非路径分隔符字符组成，会返回单个路径分隔符；其他任何情况下都不会返回以路径分隔符结尾的路径。

func Base

```
func Base(path string) string
```

Base函数返回路径的最后一个元素。在提取元素前会求掉末尾的路径分隔符。如果路径是""，会返回"."；如果路径是只有一个斜杆构成，会返回单个路径分隔符。

func Ext

```
func Ext(path string) string
```

Ext函数返回path文件扩展名。返回值是路径最后一个路径元素的最后一个'.'起始的后缀（包括'.'）。如果该元素没有'.'会返回空字符串。

func Clean

```
func Clean(path string) string
```

Clean函数通过单纯的词法操作返回和path代表同一地址的最短路径。

它会不断的依次应用如下的规则，直到不能再进行任何处理：

```
1\. 将连续的多个路径分隔符替换为单个路径分隔符
2\. 剔除每一个.路径名元素（代表当前目录）
3\. 剔除每一个路径内的..路径名元素（代表父目录）和它前面的非..路径名元素
4\. 剔除开始一个根路径的..路径名元素，即将路径开始处的"/.."替换为"/"（假设路径
```

返回的路径只有其代表一个根地址时才以路径分隔符结尾，如Unix的"/"或Windows的 `C:\` 。

如果处理的结果是空字符串，Clean会返回"."。参见http://plan9.bell-labs.com/sys/doc/lexnames.html

func EvalSymlinks

```
func EvalSymlinks(path string) (string, error)
```

EvalSymlinks函数返回path指向的符号链接（软链接）所包含的路径。如果path和返回值都是相对路径，会相对于当前目录；除非两个路径其中一个是绝对路径。

func Match

```
func Match(pattern, name string) (matched bool, err error)
```

Match returns true if name matches the shell file name pattern. The pattern syntax is:

```
pattern:
    { term }
term:
    '*'                                   匹配0或多个非路径分隔符的字符
    '?'                                   匹配1个非路径分隔符的字符
    '[' [ '^' ] { character-range } ']'   字符组（必须非空）
    c                                     匹配字符c（c != '*', '?', '
    '\\' c                                匹配字符c
character-range:
    c           匹配字符c（c != '\\', '-', ']')
    '\\' c      匹配字符c
    lo '-' hi   匹配区间[lo, hi]内的字符
```

Match要求匹配整个name字符串，而不是它的一部分。只有pattern语法错误时，会返回ErrBadPattern。

Windows系统中，不能进行转义：'\'被视为路径分隔符。

func Glob

```
func Glob(pattern string) (matches []string, err error)
```

Glob函数返回所有匹配模式匹配字符串pattern的文件或者nil（如果没有匹配的文件）。pattern的语法和Match函数相同。pattern可以描述多层的名字，如/usr/*/bin/ed（假设路径分隔符是'/'）。

type WalkFunc

```
type WalkFunc func(path string, info os.FileInfo, err error) error
```

Walk函数对每一个文件/目录都会调用WalkFunc函数类型值。调用时path参数会包含Walk的root参数作为前缀；就是说，如果Walk函数的root为"dir"，该目录下有文件"a"，将会使用"dir/a"调用walkFn参数。walkFn参数被调用时的info参数是path指定的地址（文件/目录）的文件信息，类型为os.FileInfo。

如果遍历path指定的文件或目录时出现了问题，传入的参数err会描述该问题，WalkFunc类型函数可以决定如何去处理该错误（Walk函数将不会深入该目录）；如果该函数返回一个错误，Walk函数的执行会中止；只有一个例外，如果Walk的walkFn返回值是SkipDir，将会跳过该目录的内容而Walk函数照常执行处理下一个文件。

func Walk

```
func Walk(root string, walkFn WalkFunc) error
```

Walk函数会遍历root指定的目录下的文件树，对每一个该文件树中的目录和文件都会调用walkFn，包括root自身。所有访问文件/目录时遇到的错误都会传递给walkFn过滤。文件是按词法顺序遍历的，这让输出更漂亮，但也导致处理非常大的目录时效率会降低。Walk函数不会遍历文件树中的符号链接（快捷方式）文件包含的路径。

func HasPrefix

```
func HasPrefix(p, prefix string) bool
```

HasPrefix函数出于历史兼容问题保留，不应被使用。

package reflect

```
import "reflect"
```

reflect包实现了运行时反射，允许程序操作任意类型的对象。典型用法是用静态类型interface{}保存一个值，通过调用TypeOf获取其动态类型信息，该函数返回一个Type类型值。调用ValueOf函数返回一个Value类型值，该值代表运行时的数据。Zero接受一个Type类型参数并返回一个代表该类型零值的Value类型值。

参见"The Laws of Reflection"获取go反射的介绍：http://golang.org/doc/articles/laws_of_reflection.html

Index

Examples

Constants

```
const (
    SelectSend    // case Chan <- Send
    SelectRecv    // case <-Chan:
    SelectDefault // default
)
```

type ValueError

```
type ValueError struct {
    Method string
    Kind   Kind
}
```

当一个Value类型值调用它不支持的方法时，将导致ValueError。具体情况参见各个方法。

func (*ValueError) Error

```
func (e *ValueError) Error() string
```

type Kind

```
type Kind uint
```

Kind代表Type类型值表示的具体分类。零值表示非法分类。

```
const (
    Invalid Kind = iota
    Bool
    Int
    Int8
    Int16
    Int32
    Int64
    Uint
    Uint8
    Uint16
    Uint32
    Uint64
    Uintptr
    Float32
    Float64
    Complex64
    Complex128
    Array
    Chan
    Func
    Interface
    Map
    Ptr
    Slice
    String
    Struct
    UnsafePointer
)
```

func (Kind) String

```
func (k Kind) String() string
```

type StringHeader

```
type StringHeader struct {
    Data uintptr
    Len  int
}
```

StringHeader代表一个运行时的字符串。它不保证使用的可移植性、安全性；它的实现在未来的版本里也可能会改变。而且，Data字段也不能保证它指向的数据不会被当成垃圾收集，因此程序必须维护一个独立的、类型正确的指向底层数据的指针。

type SliceHeader

```
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}
```

SliceHeader代表一个运行时的切片。它不保证使用的可移植性、安全性；它的实现在未来的版本里也可能会改变。而且，Data字段也不能保证它指向的数据不会被当成垃圾收集，因此程序必须维护一个独立的、类型正确的指向底层数据的指针。

type StructField

```
type StructField struct {
    // Name是字段的名字。PkgPath是非导出字段的包路径，对导出字段该字段为""。
    // 参见http://golang.org/ref/spec#Uniqueness_of_identifiers
    Name    string
    PkgPath string
    Type      Type      // 字段的类型
    Tag       StructTag // 字段的标签
    Offset    uintptr   // 字段在结构体中的字节偏移量
    Index     []int     // 用于Type.FieldByIndex时的索引切片
    Anonymous bool      // 是否匿名字段
}
```

StructField类型描述结构体中的一个字段的信息。

type StructTag

```
type StructTag string
```

StructTag是结构体字段的标签。

一般来说，标签字符串是（可选的）空格分隔的一连串 `key:"value"` 对。每个键都是不包含控制字符、空格、双引号、冒号的非空字符串。每个值都应被双引号括起来，使用go字符串字面语法。

Example

```
type S struct {
    F string `species:"gopher" color:"blue"`
}
s := S{}
st := reflect.TypeOf(s)
field := st.Field(0)
fmt.Println(field.Tag.Get("color"), field.Tag.Get("species"))
```

Output:

```
blue gopher
```

func (StructTag) Get

```
func (tag StructTag) Get(key string) string
```

Get方法返回标签字符串中键key对应的值。如果标签中没有该键，会返回""。如果标签不符合标准格式，Get的返回值是不确定的。

type ChanDir

```
type ChanDir int
```

ChanDir表示通道类型的方向。

```
const (
    RecvDir ChanDir             = 1 << iota // <-chan
    SendDir                                 // chan<-
    BothDir = RecvDir | SendDir             // chan
)
```

func (ChanDir) String

```
func (d ChanDir) String() string
```

type SelectDir

```
type SelectDir int
```

SelectDir描述一个SelectCase的通信方向。

type SelectCase

```
type SelectCase struct {
    Dir  SelectDir // case的方向
    Chan Value     // 使用的通道（收/发）
    Send Value     // 用于发送的值
}
```

SelectCase描述select操作中的单条case。Case的类型由通信方向Dir决定。

如果Dir是SelectDefault，该条case代表default case。Chan和Send字段必须是Value零值。

如果Dir是SelectSend，该条case代表一个发送操作。Chan字段底层必须是一个chan类型，Send的底层必须是可以直接赋值给该chan类型成员类型的类型。如果Chan是Value零值，则不管Send字段是不是零值，该条case都会被忽略。

如果Dir是SelectRecv，该条case代表一个接收操作。Chan字段底层必须是一个chan类型，而Send必须是一个Value零值。如果Chan是Value零值，该条case会被忽略，但Send字段仍需是Value零值。当该条case被执行时，接收到的值会被Select返回。

func Select

```
func Select(cases []SelectCase) (chosen int, recv Value, recvOK boo
```

Select函数执行cases切片描述的select操作。 类似go的select语句，它会阻塞直到至少一条case可以执行，从可执行的case中（伪）随机的选择一条，并执行该条case。它会返回选择执行的case的索引，以及如果执行的是接收case时，会返回接

收到的值，以及一个布尔值说明该值是否对应于通道中某次发送的值（用以区分通道关闭时接收到的零值，此时recvOK会设为false）。

type Method

```
type Method struct {
    // Name是方法名。PkgPath是非导出字段的包路径，对导出字段该字段为""。
    // 结合PkgPath和Name可以从方法集中指定一个方法。
    // 参见http://golang.org/ref/spec#Uniqueness_of_identifiers
    Name    string
    PkgPath string
    Type  Type  // 方法类型
    Func  Value // 方法的值
    Index int   // 用于Type.Method的索引
}
```

Method代表一个方法。

type Type

```
type Type interface {
    // Kind返回该接口的具体分类
    Kind() Kind
    // Name返回该类型在自身包内的类型名，如果是未命名类型会返回""
    Name() string
    // PkgPath返回类型的包路径，即明确指定包的import路径，如"encoding/bas
    // 如果类型为内建类型(string, error)或未命名类型(*T, struct{}, []int
    PkgPath() string
    // 返回类型的字符串表示。该字符串可能会使用短包名（如用base64代替"encodi
    // 也不保证每个类型的字符串表示不同。如果要比较两个类型是否相等，请直接用T
    String() string
    // 返回要保存一个该类型的值需要多少字节；类似unsafe.Sizeof
    Size() uintptr
    // 返回当从内存中申请一个该类型值时，会对齐的字节数
    Align() int
    // 返回当该类型作为结构体的字段时，会对齐的字节数
    FieldAlign() int
    // 如果该类型实现了u代表的接口，会返回真
    Implements(u Type) bool
    // 如果该类型的值可以直接赋值给u代表的类型，返回真
    AssignableTo(u Type) bool
    // 如该类型的值可以转换为u代表的类型，返回真
    ConvertibleTo(u Type) bool
    // 返回该类型的字位数。如果该类型的Kind不是Int、Uint、Float或Complex，
    Bits() int
    // 返回array类型的长度，如非数组类型将panic
    Len() int
```

```
    // 返回该类型的元素类型，如果该类型的Kind不是Array、Chan、Map、Ptr或Sl
    Elem() Type
    // 返回map类型的键的类型。如非映射类型将panic
    Key() Type
    // 返回一个channel类型的方向，如非通道类型将会panic
    ChanDir() ChanDir
    // 返回struct类型的字段数（匿名字段算作一个字段），如非结构体类型将panic
    NumField() int
    // 返回struct类型的第i个字段的类型，如非结构体或者i不在[0, NumField())
    Field(i int) StructField
    // 返回索引序列指定的嵌套字段的类型，
    // 等价于用索引中每个值链式调用本方法，如非结构体将会panic
    FieldByIndex(index []int) StructField
    // 返回该类型名为name的字段（会查找匿名字段及其子字段），
    // 布尔值说明是否找到，如非结构体将panic
    FieldByName(name string) (StructField, bool)
    // 返回该类型第一个字段名满足函数match的字段，布尔值说明是否找到，如非结构
    FieldByNameFunc(match func(string) bool) (StructField, bool)
    // 如果函数类型的最后一个输入参数是"..."形式的参数，IsVariadic返回真
    // 如果这样，t.In(t.NumIn() - 1)返回参数的隐式的实际类型（声明类型的切片
    // 如非函数类型将panic
    IsVariadic() bool
    // 返回func类型的参数个数，如果不是函数，将会panic
    NumIn() int
    // 返回func类型的第i个参数的类型，如非函数或者i不在[0, NumIn())内将会pa
    In(i int) Type
    // 返回func类型的返回值个数，如果不是函数，将会panic
    NumOut() int
    // 返回func类型的第i个返回值的类型，如非函数或者i不在[0, NumOut())内将会
    Out(i int) Type
    // 返回该类型的方法集中方法的数目
    // 匿名字段的方法会被计算；主体类型的方法会屏蔽匿名字段的同名方法；
    // 匿名字段导致的歧义方法会滤除
    NumMethod() int
    // 返回该类型方法集中的第i个方法，i不在[0, NumMethod())范围内时，将导致
    // 对非接口类型T或*T，返回值的Type字段和Func字段描述方法的未绑定函数状态
    // 对接口类型，返回值的Type字段描述方法的签名，Func字段为nil
    Method(int) Method
    // 根据方法名返回该类型方法集中的方法，使用一个布尔值说明是否发现该方法
    // 对非接口类型T或*T，返回值的Type字段和Func字段描述方法的未绑定函数状态
    // 对接口类型，返回值的Type字段描述方法的签名，Func字段为nil
    MethodByName(string) (Method, bool)
    // 内含隐藏或非导出方法
}
```

Type类型用来表示一个go类型。

不是所有go类型的Type值都能使用所有方法。请参见每个方法的文档获取使用限制。在调用有分类限定的方法时，应先使用Kind方法获知类型的分类。调用该分类不支持的方法会导致运行时的panic。

func TypeOf

```
func TypeOf(i interface{}) Type
```

TypeOf返回接口中保存的值的类型，TypeOf(nil)会返回nil。

func PtrTo

```
func PtrTo(t Type) Type
```

PtrTo返回类型t的指针的类型。

func SliceOf

```
func SliceOf(t Type) Type
```

SliceOf返回类型t的切片的类型。

func MapOf

```
func MapOf(key, elem Type) Type
```

MapOf返回一个键类型为key，值类型为elem的映射类型。如果key代表的类型不是合法的映射键类型（即它未实现go的==操作符），本函数会panic。

func ChanOf

```
func ChanOf(dir ChanDir, t Type) Type
```

ChanOf返回元素类型为t、方向为dir的通道类型。运行时GC强制将通道的元素类型的大小限定为64kb。如果t的尺寸大于或等于该限制，本函数将会panic。

type Value

```
type Value struct {
    // 内含隐藏或非导出字段
}
```

Value为go值提供了反射接口。

不是所有go类型值的Value表示都能使用所有方法。请参见每个方法的文档获取使用限制。在调用有分类限定的方法时，应先使用Kind方法获知该值的分类。调用该分类不支持的方法会导致运行时的panic。

Value类型的零值表示不持有某个值。零值的IsValid方法返回false，其Kind方法返回Invalid，而String方法返回"<invalid Value>"，所有其它方法都会panic。绝大多数函数和方法都永远不返回Value零值。如果某个函数/方法返回了非法的Value，它的文档必须显式的说明具体情况。

如果某个go类型值可以安全的用于多线程并发操作，它的Value表示也可以安全的用于并发。

func ValueOf

```
func ValueOf(i interface{}) Value
```

ValueOf返回一个初始化为i接口保管的具体值的Value，ValueOf(nil)返回Value零值。

func Zero

```
func Zero(typ Type) Value
```

Zero返回一个持有类型typ的零值的Value。注意持有零值的Value和Value零值是两回事。Value零值表示不持有任何值。例如Zero(TypeOf(42))返回一个Kind为Int、值为0的Value。Zero的返回值不能设置也不会寻址。

func New

```
func New(typ Type) Value
```

New返回一个Value类型值，该值持有一个指向类型为typ的新申请的零值的指针，返回值的Type为PtrTo(typ)。

func NewAt

```
func NewAt(typ Type, p unsafe.Pointer) Value
```

NewAt返回一个Value类型值，该值持有一个指向类型为typ、地址为p的值的指针。

func Indirect

```
func Indirect(v Value) Value
```

返回持有v持有的指针指向的值的Value。如果v持有nil指针，会返回Value零值；如果v不持有指针，会返回v。

func MakeSlice

```
func MakeSlice(typ Type, len, cap int) Value
```

MakeSlice创建一个新申请的元素类型为typ，长度len容量cap的切片类型的Value值。

func MakeMap

```
func MakeMap(typ Type) Value
```

MakeMap创建一个特定映射类型的Value值。

func MakeChan

```
func MakeChan(typ Type, buffer int) Value
```

MakeChan创建一个元素类型为typ、有buffer个缓存的通道类型的Value值。

func MakeFunc

```
func MakeFunc(typ Type, fn func(args []Value) (results []Value)) Va
```

MakeFunc返回一个具有给定类型、包装函数fn的函数的Value封装。当被调用时，该函数会：

```
- 将提供给它的参数转化为Value切片
- 执行results := fn(args)
- 将results中每一个result依次排列作为返回值
```

函数fn的实现可以假设参数Value切片匹配typ类型指定的参数数目和类型。如果typ表示一个可变参数函数类型，参数切片中最后一个Value本身必须是一个包含所有可变参数的切片。fn返回的结果Value切片也必须匹配typ类型指定的结果数目和类型。

Value.Call方法允许程序员使用Value调用一个有类型约束的函数；反过来，MakeFunc方法允许程序员使用Value实现一个有类型约束的函数。

下例是一个用MakeFunc创建一个生成不同参数类型的swap函数的代码及其说明。

Example

```
// swap is the implementation passed to MakeFunc.
// It must work in terms of reflect.Values so that it is possible
// to write code without knowing beforehand what the types
// will be.
swap := func(in []reflect.Value) []reflect.Value {
    return []reflect.Value{in[1], in[0]}
}
// makeSwap expects fptr to be a pointer to a nil function.
// It sets that pointer to a new function created with MakeFunc.
// When the function is invoked, reflect turns the arguments
// into Values, calls swap, and then turns swap's result slice
// into the values returned by the new function.
makeSwap := func(fptr interface{}) {
    // fptr is a pointer to a function.
    // Obtain the function value itself (likely nil) as a reflect.\
    // so that we can query its type and then set the value.
    fn := reflect.ValueOf(fptr).Elem()
    // Make a function of the right type.
    v := reflect.MakeFunc(fn.Type(), swap)
    // Assign it to the value fn represents.
    fn.Set(v)
}
// Make and call a swap function for ints.
var intSwap func(int, int) (int, int)
makeSwap(&intSwap)
fmt.Println(intSwap(0, 1))
// Make and call a swap function for float64s.
var floatSwap func(float64, float64) (float64, float64)
makeSwap(&floatSwap)
fmt.Println(floatSwap(2.72, 3.14))
```

Output:

```
1 0
3.14 2.72
```

func Append

```
func Append(s Value, x ...Value) Value
```

向切片类型的Value值s中添加一系列值，x等Value值持有的值必须能直接赋值给s持有的切片的元素类型。

func AppendSlice

```
func AppendSlice(s, t Value) Value
```

类似Append函数，但接受一个切片类型的Value值。将切片t的每一个值添加到s。

func (Value) IsValid

```
func (v Value) IsValid() bool
```

IsValid返回v是否持有一个值。如果v是Value零值会返回假，此时v除了IsValid、String、Kind之外的方法都会导致panic。绝大多数函数和方法都永远不返回Value零值。如果某个函数/方法返回了非法的Value，它的文档必须显式的说明具体情况。

func (Value) IsNil

```
func (v Value) IsNil() bool
```

IsNil报告v持有的值是否为nil。v持有的值的分类必须是通道、函数、接口、映射、指针、切片之一；否则IsNil函数会导致panic。注意IsNil并不总是等价于go语言中值与nil的常规比较。例如：如果v是通过使用某个值为nil的接口调用ValueOf函数创建的，v.IsNil()返回真，但是如果v是Value零值，会panic。

func (Value) Kind

```
func (v Value) Kind() Kind
```

Kind返回v持有的值的分类，如果v是Value零值，返回值为Invalid

func (Value) Type

```
func (v Value) Type() Type
```

返回v持有的值的类型的Type表示。

func (Value) Convert

```
func (v Value) Convert(t Type) Value
```

Convert将v持有的值转换为类型为t的值，并返回该值的Value封装。如果go转换规则不支持这种转换，会panic。

func (Value) Elem

```
func (v Value) Elem() Value
```

Elem返回v持有的接口保管的值的Value封装，或者v持有的指针指向的值的Value封装。如果v的Kind不是Interface或Ptr会panic；如果v持有的值为nil，会返回Value零值。

func (Value) Bool

```
func (v Value) Bool() bool
```

返回v持有的布尔值，如果v的Kind不是Bool会panic

func (Value) Int

```
func (v Value) Int() int64
```

返回v持有的有符号整数（表示为int64），如果v的Kind不是Int、Int8、Int16、Int32、Int64会panic

func (Value) OverflowInt

```
func (v Value) OverflowInt(x int64) bool
```

如果v持有值的类型不能无溢出的表示x，会返回真。如果v的Kind不是Int、Int8、Int16、Int32、Int64会panic

func (Value) Uint

```
func (v Value) Uint() uint64
```

返回v持有的无符号整数（表示为uint64），如v的Kind不是Uint、Uintptr、Uint8、Uint16、Uint32、Uint64会panic

func (Value) OverflowUint

```
func (v Value) OverflowUint(x uint64) bool
```

如果v持有值的类型不能无溢出的表示x，会返回真。如果v的Kind不是Uint、Uintptr、Uint8、Uint16、Uint32、Uint64会panic

func (Value) Float

```
func (v Value) Float() float64
```

返回v持有的浮点数（表示为float64），如果v的Kind不是Float32、Float64会panic

func (Value) OverflowFloat

```
func (v Value) OverflowFloat(x float64) bool
```

如果v持有值的类型不能无溢出的表示x，会返回真。如果v的Kind不是Float32、Float64会panic

func (Value) Complex

```
func (v Value) Complex() complex128
```

返回v持有的复数（表示为complex64），如果v的Kind不是Complex64、Complex128会panic

func (Value) OverflowComplex

```
func (v Value) OverflowComplex(x complex128) bool
```

如果v持有值的类型不能无溢出的表示x，会返回真。如果v的Kind不是Complex64、Complex128会panic

func (Value) Pointer

```
func (v Value) Pointer() uintptr
```

将v持有的值作为一个指针返回。本方法返回值不是unsafe.Pointer类型，以避免程序员不显式导入unsafe包却得到unsafe.Pointer类型表示的指针。如果v的Kind不是Chan、Func、Map、Ptr、Slice或UnsafePointer会panic。

如果v的Kind是Func，返回值是底层代码的指针，但并不足以用于区分不同的函数；只能保证当且仅当v持有函数类型零值nil时，返回值为0。

如果v的Kind是Slice，返回值是指向切片第一个元素的指针。如果持有的切片为nil，返回值为0；如果持有的切片没有元素但不是nil，返回值不会是0。

func (Value) Bytes

```
func (v Value) Bytes() []byte
```

返回v持有的[]byte类型值。如果v持有的值的类型不是[]byte会panic。

func (Value) String

```
func (v Value) String() string
```

返回v持有的值的字符串表示。因为go的String方法的惯例，Value的String方法比较特别。和其他获取v持有值的方法不同：v的Kind是String时，返回该字符串；v的Kind不是String时也不会panic而是返回格式为"<T value>"的字符串，其中T是v持有值的类型。

func (Value) InterfaceData

```
func (v Value) InterfaceData() [2]uintptr
```

返回v持有的接口类型值的数据。如果v的Kind不是Interface会panic

func (Value) Slice

```
func (v Value) Slice(i, j int) Value
```

返回v[i:j]（v持有的切片的子切片的Value封装）；如果v的Kind不是Array、Slice或String会panic。如果v是一个不可寻址的数组，或者索引出界，也会panic

func (Value) Slice3

```
func (v Value) Slice3(i, j, k int) Value
```

是Slice的3参数版本，返回v[i:j:k] ；如果v的Kind不是Array、Slice或String会panic。如果v是一个不可寻址的数组，或者索引出界，也会panic。

func (Value) Cap

```
func (v Value) Cap() int
```

返回v持有值的容量，如果v的Kind不是Array、Chan、Slice会panic

func (Value) Len

```
func (v Value) Len() int
```

返回v持有值的长度，如果v的Kind不是Array、Chan、Slice、Map、String会panic

func (Value) Index

```
func (v Value) Index(i int) Value
```

返回v持有值的第i个元素。如果v的Kind不是Array、Chan、Slice、String，或者i出界，会panic

func (Value) MapIndex

```
func (v Value) MapIndex(key Value) Value
```

返回v持有值里key持有值为键对应的值的Value封装。如果v的Kind不是Map会panic。如果未找到对应值或者v持有值是nil映射，会返回Value零值。key的持有值必须可以直接赋值给v持有值类型的键类型。

func (Value) MapKeys

```
func (v Value) MapKeys() []Value
```

返回一个包含v持有值中所有键的Value封装的切片，该切片未排序。如果v的Kind不是Map会panic。如果v持有值是nil，返回空切片（非nil）。

func (Value) NumField

```
func (v Value) NumField() int
```

返回v持有的结构体类型值的字段数，如果v的Kind不是Struct会panic

func (Value) Field

```
func (v Value) Field(i int) Value
```

返回结构体的第i个字段（的Value封装）。如果v的Kind不是Struct或i出界会panic

func (Value) FieldByIndex

```
func (v Value) FieldByIndex(index []int) Value
```

返回索引序列指定的嵌套字段的Value表示，等价于用索引中的值链式调用本方法，如v的Kind非Struct将会panic

func (Value) FieldByName

```
func (v Value) FieldByName(name string) Value
```

返回该类型名为name的字段（的Value封装）（会查找匿名字段及其子字段），如果v的Kind不是Struct会panic；如果未找到会返回Value零值。

func (Value) FieldByNameFunc

```
func (v Value) FieldByNameFunc(match func(string) bool) Value
```

返回该类型第一个字段名满足match的字段（的Value封装）（会查找匿名字段及其子字段），如果v的Kind不是Struct会panic；如果未找到会返回Value零值。

func (Value) Recv

```
func (v Value) Recv() (x Value, ok bool)
```

方法从v持有的通道接收并返回一个值（的Value封装）。如果v的Kind不是Chan会panic。方法会阻塞直到获取到值。如果返回值x对应于某个发送到v持有的通道的值，ok为真；如果因为通道关闭而返回，x为Value零值而ok为假。

func (Value) TryRecv

```
func (v Value) TryRecv() (x Value, ok bool)
```

TryRecv尝试从v持有的通道接收一个值，但不会阻塞。如果v的Kind不是Chan会panic。如果方法成功接收到一个值，会返回该值（的Value封装）和true；如果不能无阻塞的接收到值，返回Value零值和false；如果因为通道关闭而返回，返回值x是持有通道元素类型的零值的Value和false。

func (Value) Send

```
func (v Value) Send(x Value)
```

方法向v持有的通道发送x持有的值。如果v的Kind不是Chan，或者x的持有值不能直接赋值给v持有通道的元素类型，会panic。

func (Value) TrySend

```
func (v Value) TrySend(x Value) bool
```

TrySend尝试向v持有的通道发送x持有的值，但不会阻塞。如果v的Kind不是Chan会panic。如果成功发送会返回真，否则返回假。x的持有值必须可以直接赋值给v持有通道的元素类型。

func (Value) Close

```
func (v Value) Close()
```

关闭v持有的通道，如果v的Kind不是Chan会panic

func (Value) Call

```
func (v Value) Call(in []Value) []Value
```

Call方法使用输入的参数in调用v持有的函数。例如，如果len(in) == 3，v.Call(in)代表调用v(in[0], in[1], in[2])（其中Value值表示其持有值）。如果v的Kind不是Func会panic。它返回函数所有输出结果的Value封装的切片。和go代码一样，每一个输入实参的持有值都必须可以直接赋值给函数对应输入参数的类型。如果v持有值是可变参数函数，Call方法会自行创建一个代表可变参数的切片，将对应可变参数的值都拷贝到里面。

func (Value) CallSlice

```
func (v Value) CallSlice(in []Value) []Value
```

CallSlice调用v持有的可变参数函数，会将切片类型的in[len(in)-1]（的成员）分配给v的最后的可变参数。例如，如果len(in) == 3，v.Call(in)代表调用v(in[0], in[1], in[2])（其中Value值表示其持有值，可变参数函数的可变参数位置提供一个切片并跟三个点号代表"解切片"）。如果v的Kind不是Func或者v的持有值不是可变参数函数，会panic。它返回函数所有输出结果的Value封装的切片。和go代码一样，每一个输入实参的持有值都必须可以直接赋值给函数对应输入参数的类型。

func (Value) NumMethod

```
func (v Value) NumMethod() int
```

返回v持有值的方法集的方法数目。

func (Value) Method

```
func (v Value) Method(i int) Value
```

返回v持有值类型的第i个方法的已绑定（到v的持有值的）状态的函数形式的Value封装。返回值调用Call方法时不应包含接收者；返回值持有的函数总是使用v的持有者作为接收者（即第一个参数）。如果i出界，或者v的持有值是接口类型的零值

（nil），会panic。

func (Value) MethodByName

```
func (v Value) MethodByName(name string) Value
```

返回v的名为name的方法的已绑定（到v的持有值的）状态的函数形式的Value封装。返回值调用Call方法时不应包含接收者；返回值持有的函数总是使用v的持有者作为接收者（即第一个参数）。如果未找到该方法，会返回一个Value零值。

func (Value) CanAddr

```
func (v Value) CanAddr() bool
```

返回是否可以获取v持有值的指针。可以获取指针的值被称为可寻址的。如果一个值是切片或可寻址数组的元素、可寻址结构体的字段、或从指针解引用得到的，该值即为可寻址的。

func (Value) Addr

```
func (v Value) Addr() Value
```

函数返回一个持有指向v持有者的指针的Value封装。如果v.CanAddr()返回假，调用本方法会panic。Addr一般用于获取结构体字段的指针或者切片的元素（的Value封装）以便调用需要指针类型接收者的方法。

func (Value) UnsafeAddr

```
func (v Value) UnsafeAddr() uintptr
```

返回指向v持有数据的地址的指针（表示为uintptr）以用作高级用途，如果v不可寻址会panic。

func (Value) CanInterface

```
func (v Value) CanInterface() bool
```

如果CanInterface返回真，v可以不导致panic的调用Interface方法。

func (Value) Interface

```
func (v Value) Interface() (i interface{})
```

本方法返回v当前持有的值（表示为/保管在interface{}类型），等价于：

```
var i interface{} = (v's underlying value)
```

如果v是通过访问非导出结构体字段获取的，会导致panic。

func (Value) CanSet

```
func (v Value) CanSet() bool
```

如果v持有的值可以被修改，CanSet就会返回真。只有一个Value持有值可以被寻址同时又不是来自非导出字段时，它才可以被修改。如果CanSet返回假，调用Set或任何限定类型的设置函数（如SetBool、SetInt64）都会panic。

func (Value) SetBool

```
func (v Value) SetBool(x bool)
```

设置v的持有值。如果v的Kind不是Bool或者v.CanSet()返回假，会panic。

func (Value) SetInt

```
func (v Value) SetInt(x int64)
```

设置v的持有值。如果v的Kind不是Int、Int8、Int16、Int32、Int64之一或者v.CanSet()返回假，会panic。

func (Value) SetUint

```
func (v Value) SetUint(x uint64)
```

设置v的持有值。如果v的Kind不是Uint、Uintptr、Uint8、Uint16、Uint32、Uint64或者v.CanSet()返回假，会panic。

func (Value) SetFloat

```
func (v Value) SetFloat(x float64)
```

设置v的持有值。如果v的Kind不是Float32、Float64或者v.CanSet()返回假，会panic。

func (Value) SetComplex

```
func (v Value) SetComplex(x complex128)
```

设置v的持有值。如果v的Kind不是Complex64、Complex128或者v.CanSet()返回假，会panic。

func (Value) SetBytes

```
func (v Value) SetBytes(x []byte)
```

设置v的持有值。如果v持有值不是[]byte类型或者v.CanSet()返回假，会panic。

func (Value) SetString

```
func (v Value) SetString(x string)
```

设置v的持有值。如果v的Kind不是String或者v.CanSet()返回假，会panic。

func (Value) SetPointer

```
func (v Value) SetPointer(x unsafe.Pointer)
```

设置v的持有值。如果v的Kind不是UnsafePointer或者v.CanSet()返回假，会panic。

func (Value) SetCap

```
func (v Value) SetCap(n int)
```

设定v持有值的容量。如果v的Kind不是Slice或者n出界（小于长度或超出容量），将导致panic

func (Value) SetLen

```
func (v Value) SetLen(n int)
```

设定v持有值的长度。如果v的Kind不是Slice或者n出界（小于零或超出容量），将导致panic

func (Value) SetMapIndex

```
func (v Value) SetMapIndex(key, val Value)
```

用来给v的映射类型持有值添加/修改键值对，如果val是Value零值，则是删除键值对。如果v的Kind不是Map，或者v的持有值是nil，将会panic。key的持有值必须可以直接赋值给v持有值类型的键类型。val的持有值必须可以直接赋值给v持有值类型的值类型。

func (Value) Set

```
func (v Value) Set(x Value)
```

将v的持有值修改为x的持有值。如果v.CanSet()返回假，会panic。x的持有值必须能直接赋给v持有值的类型。

func Copy

```
func Copy(dst, src Value) int
```

将src中的值拷贝到dst，直到src被耗尽或者dst被装满，要求这二者都是slice或array，且元素类型相同。

func DeepEqual

```
func DeepEqual(a1, a2 interface{}) bool
```

用来判断两个值是否深度一致：除了类型相同；在可以时（主要是基本类型）会使用==；但还会比较array、slice的成员，map的键值对，结构体字段进行深入比对。map的键值对，对键只使用==，但值会继续往深层比对。DeepEqual函数可以正确处理循环的类型。函数类型只有都会nil时才相等；空切片不等于nil切片；还会考虑array、slice的长度、map键值对数。

Bugs

☞FieldByName及相关的函数会将名称相同的结构体字段视为相同的字段，即使它们是来自不同包的非导出字段。这导致如果结构体类型t包含多个名为x的字段的话（被不同的包嵌入）t.FieldByName("x")的行为没有良好的定义。FieldByName可能会返回其中一个名为x的字段，也可能报告说没有这个字段。细节参见golang.org/issue/4876。

package regexp

```
import "regexp"
```

regexp包实现了正则表达式搜索。

正则表达式采用RE2语法（除了\c、\C），和Perl、Python等语言的正则基本一致。

参见http://code.google.com/p/re2/wiki/Syntax。

Syntax

本包采用的正则表达式语法，默认采用perl标志。某些语法可以通过切换解析时的标志来关闭。

单字符：

```
        .              任意字符（标志s==true时还包括换行符）
        [xyz]          字符族
        [^xyz]         反向字符族
        \d             Perl预定义字符族
        \D             反向Perl预定义字符族
        [:alpha:]      ASCII字符族
        [:^alpha:]     反向ASCII字符族
        \pN            Unicode字符族（单字符名），参见unicode包
        \PN            反向Unicode字符族（单字符名）
        \p{Greek}      Unicode字符族（完整字符名）
        \P{Greek}      反向Unicode字符族（完整字符名）
```

结合：

```
        xy             匹配x后接着匹配y
        x|y            匹配x或y（优先匹配x）
```

重复：

```
        x*              重复>=0次匹配x，越多越好（优先重复匹配x）
        x+              重复>=1次匹配x，越多越好（优先重复匹配x）
        x?              0或1次匹配x，优先1次
        x{n,m}          n到m次匹配x，越多越好（优先重复匹配x）
        x{n,}           重复>=n次匹配x，越多越好（优先重复匹配x）
        x{n}            重复n次匹配x
        x*?             重复>=0次匹配x，越少越好（优先跳出重复）
        x+?             重复>=1次匹配x，越少越好（优先跳出重复）
        x??             0或1次匹配x，优先0次
        x{n,m}?         n到m次匹配x，越少越好（优先跳出重复）
        x{n,}?          重复>=n次匹配x，越少越好（优先跳出重复）
        x{n}?           重复n次匹配x
```

实现的限制：计数格式x{n}等（不包括x*等格式）中n最大值1000。负数或者显式出现的过大的值会导致解析错误，返回ErrInvalidRepeatSize。

分组：

```
        (re)            编号的捕获分组
        (?P<name>re)    命名并编号的捕获分组
        (?:re)          不捕获的分组
        (?flags)        设置当前所在分组的标志，不捕获也不匹配
        (?flags:re)     设置re段的标志，不捕获的分组
```

标志的语法为xyz（设置）、-xyz（清楚）、xy-z（设置xy，清楚z），标志如下：

```
        I               大小写敏感（默认关闭）
        m               ^和$在匹配文本开始和结尾之外，还可以匹配行首和行尾（
        s               让.可以匹配\n（默认关闭）
        U               非贪婪的：交换x*和x*?、x+和x+?……的含义（默认关闭）
```

边界匹配：

```
        ^               匹配文本开始，标志m为真时，还匹配行首
        $               匹配文本结尾，标志m为真时，还匹配行尾
        \A              匹配文本开始
        \b              单词边界（一边字符属于\w，另一边为文首、文尾、行首、
        \B              非单词边界
        \z              匹配文本结尾
```

转义序列：

```
        \a              响铃符（\007）
        \f              换纸符（\014）
        \t              水平制表符（\011）
        \n              换行符（\012）
        \r              回车符（\015）
        \v              垂直制表符（\013）
        \123            八进制表示的字符码（最多三个数字）
        \x7F            十六进制表示的字符码（必须两个数字）
        \x{10FFFF}      十六进制表示的字符码
        \*              字面值'*'
        \Q...\E         反斜线后面的字符的字面值
```

字符族（预定义字符族之外，方括号内部）的语法：

```
        x               单个字符
        A-Z             字符范围（方括号内部才可以用）
        \d              Perl字符族
        [:foo:]         ASCII字符族
        \pF             单字符名的Unicode字符族
        \p{Foo}         完整字符名的Unicode字符族
```

预定义字符族作为字符族的元素：

```
        [\d]            == \d
        [^\d]           == \D
        [\D]            == \D
        [^\D]           == \d
        [[:name:]]      == [:name:]
        [^[:name:]]     == [:^name:]
        [\p{Name}]      == \p{Name}
        [^\p{Name}]     == \P{Name}
```

Perl字符族：

```
        \d              == [0-9]
        \D              == [^0-9]
        \s              == [\t\n\f\r ]
        \S              == [^\t\n\f\r ]
        \w              == [0-9A-Za-z_]
        \W              == [^0-9A-Za-z_]
```

ASCII字符族：

```
        [:alnum:]      == [0-9A-Za-z]
        [:alpha:]      == [A-Za-z]
        [:ascii:]      == [\x00-\x7F]
        [:blank:]      == [\t ]
        [:cntrl:]      == [\x00-\x1F\x7F]
        [:digit:]      == [0-9]
        [:graph:]      == [!-~] == [A-Za-z0-9!"#$%&'()*+,\-./:;<=>?
        [:lower:]      == [a-z]
        [:print:]      == [ -~] == [ [:graph:]]
        [:punct:]      == [!-/:-@[-`{-~]
        [:space:]      == [\t\n\v\f\r ]
        [:upper:]      == [A-Z]
        [:word:]       == [0-9A-Za-z_]
        [:xdigit:]     == [0-9A-Fa-f]
```

本包的正则表达式保证搜索复杂度为O(n)，其中n为输入的长度。这一点很多其他开源实现是无法保证的。参见：

```
http://swtch.com/~rsc/regexp/regexp1.html
```

或其他关于自动机理论的书籍。

所有的字符都被视为utf-8编码的码值。

Regexp类型提供了多达16个方法，用于匹配正则表达式并获取匹配的结果。它们的名字满足如下正则表达式：

```
Find(All)?(String)?(Submatch)?(Index)?
```

如果'All'出现了，该方法会返回输入中所有互不重叠的匹配结果。如果一个匹配结果的前后（没有间隔字符）存在长度为0的成功匹配，该空匹配会被忽略。包含All的方法会要求一个额外的整数参数n，如果n>=0，方法会返回最多前n个匹配结果。

如果'String'出现了，匹配对象为字符串，否则应该是[]byte类型，返回值和匹配对象的类型是对应的。

如果'Submatch'出现了，返回值是表示正则表达式中成功的组匹配（子匹配/次级匹配）的切片。组匹配是正则表达式内部的括号包围的次级表达式（也被称为“捕获分组”），从左到右按左括号的顺序编号。，索引0的组匹配为完整表达式的匹配结果，1为第一个分组的匹配结果，依次类推。

如果'Index'出现了，匹配/分组匹配会用输入流的字节索引对表示result[2*n:2*n+1]表示第n个分组匹配的的匹配结果。如果没有'Index'，匹配结果表示为匹配到的文本。如果索引为负数，表示分组匹配没有匹配到输入流中的文本。

方法集也有一个用于从RuneReader中读取文本进行匹配的子集：

```
MatchReader, FindReaderIndex, FindReaderSubmatchIndex
```

该子集可能会增加。注意正则表达式匹配可能需要检验匹配结果前后的文本，因此从RuneReader匹配文本的方法很可能会读取到远远超出返回的结果所在的位置。

（另有几个其他方法不满足该方法模式的）

Example

```
// Compile the expression once, usually at init time.
// Use raw strings to avoid having to quote the backslashes.
var validID = regexp.MustCompile(`^[a-z]+\[[0-9]+\]$`)
fmt.Println(validID.MatchString("adam[23]"))
fmt.Println(validID.MatchString("eve[7]"))
fmt.Println(validID.MatchString("Job[48]"))
fmt.Println(validID.MatchString("snakey"))
```

Output:

```
true
true
false
false
```

Index

Examples

func QuoteMeta

```
func QuoteMeta(s string) string
```

QuoteMeta返回将s中所有正则表达式元字符都进行转义后字符串。该字符串可以用在正则表达式中匹配字面值s。例如，QuoteMeta(`[foo]`)会返回 `\[foo\]` 。

func Match

```
func Match(pattern string, b []byte) (matched bool, err error)
```

Match检查b中是否存在匹配pattern的子序列。更复杂的用法请使用Compile函数和Regexp对象。

func MatchString

```
func MatchString(pattern string, s string) (matched bool, err erro
```

MatchString类似Match，但匹配对象是字符串。

Example

```
matched, err := regexp.MatchString("foo.*", "seafood")
fmt.Println(matched, err)
matched, err = regexp.MatchString("bar.*", "seafood")
fmt.Println(matched, err)
matched, err = regexp.MatchString("a(b", "seafood")
fmt.Println(matched, err)
```

Output:

```
true <nil>
false <nil>
false error parsing regexp: missing closing ): `a(b`
```

func MatchReader

```
func MatchReader(pattern string, r io.RuneReader) (matched bool, e
```

MatchReader类似Match，但匹配对象是io.RuneReader。

type Regexp

```
type Regexp struct {
    // 内含隐藏或非导出字段
}
```

Regexp代表一个编译好的正则表达式。Regexp可以被多线程安全地同时使用。

func Compile

```
func Compile(expr string) (*Regexp, error)
```

Compile解析并返回一个正则表达式。如果成功返回，该Regexp就可用于匹配文本。

在匹配文本时，该正则表达式会尽可能早的开始匹配，并且在匹配过程中选择回溯搜索到的第一个匹配结果。这种模式被称为“leftmost-first”，Perl、Python和其他实现都采用了这种模式，但本包的实现没有回溯的损耗。对POSIX的“leftmost-longest”模式，参见CompilePOSIX。

func CompilePOSIX

```
func CompilePOSIX(expr string) (*Regexp, error)
```

类似Compile但会将语法约束到POSIX ERE（egrep）语法，并将匹配模式设置为leftmost-longest。

在匹配文本时，该正则表达式会尽可能早的开始匹配，并且在匹配过程中选择搜索到的最长的匹配结果。这种模式被称为“leftmost-longest”，POSIX采用了这种模式（早期正则的DFA自动机模式）。

然而，可能会有多个“leftmost-longest”匹配，每个都有不同的组匹配状态，本包在这里和POSIX不同。在所有可能的“leftmost-longest”匹配里，本包选择回溯搜索时第一个找到的，而POSIX会选择候选结果中第一个组匹配最长的（可能有多个），然后再从中选出第二个组匹配最长的，依次类推。POSIX规则计算困难，甚至没有良好定义。

参见http://swtch.com/~rsc/regexp/regexp2.html#posix获取细节。

func MustCompile

```
func MustCompile(str string) *Regexp
```

MustCompile类似Compile但会在解析失败时panic，主要用于全局正则表达式变量的安全初始化。

func MustCompilePOSIX

```
func MustCompilePOSIX(str string) *Regexp
```

MustCompilePOSIX类似CompilePOSIX但会在解析失败时panic，主要用于全局正则表达式变量的安全初始化。

func (*Regexp) String

```
func (re *Regexp) String() string
```

String返回用于编译成正则表达式的字符串。

func (*Regexp) LiteralPrefix

```
func (re *Regexp) LiteralPrefix() (prefix string, complete bool)
```

LiteralPrefix返回一个字符串字面值prefix，任何匹配本正则表达式的字符串都会以prefix起始。 如果该字符串字面值包含整个正则表达式，返回值complete会设为真。

func (*Regexp) NumSubexp

```
func (re *Regexp) NumSubexp() int
```

NumSubexp返回该正则表达式中捕获分组的数量。

func (*Regexp) SubexpNames

```
func (re *Regexp) SubexpNames() []string
```

SubexpNames返回该正则表达式中捕获分组的名字。第一个分组的名字是names[1]，因此，如果m是一个组匹配切片，m[i]的名字是SubexpNames()[i]。因为整个正则表达式是无法被命名的，names[0]必然是空字符串。该切片不应被修改。

Example

```
re := regexp.MustCompile("(?P<first>[a-zA-Z]+) (?P<last>[a-zA-Z]+)'
fmt.Println(re.MatchString("Alan Turing"))
fmt.Printf("%q\n", re.SubexpNames())
reversed := fmt.Sprintf("${%s} ${%s}", re.SubexpNames()[2], re.Sube
fmt.Println(reversed)
fmt.Println(re.ReplaceAllString("Alan Turing", reversed))
```

Output:

```
true
["" "first" "last"]
${last} ${first}
Turing Alan
```

func (*Regexp) Longest

```
func (re *Regexp) Longest()
```

Longest让正则表达式在之后的搜索中都采用"leftmost-longest"模式。在匹配文本时，该正则表达式会尽可能早的开始匹配，并且在匹配过程中选择搜索到的最长的匹配结果。

func (*Regexp) Match

```
func (re *Regexp) Match(b []byte) bool
```

Match检查b中是否存在匹配pattern的子序列。

func (*Regexp) MatchString

```
func (re *Regexp) MatchString(s string) bool
```

MatchString类似Match，但匹配对象是字符串。

func (*Regexp) MatchReader

```
func (re *Regexp) MatchReader(r io.RuneReader) bool
```

MatchReader类似Match，但匹配对象是io.RuneReader。

func (*Regexp) Find

```
func (re *Regexp) Find(b []byte) []byte
```

Find返回保管正则表达式re在b中的最左侧的一个匹配结果的[]byte切片。如果没有匹配到，会返回nil。

func (*Regexp) FindString

```
func (re *Regexp) FindString(s string) string
```

Find返回保管正则表达式re在b中的最左侧的一个匹配结果的字符串。如果没有匹配到，会返回""；但如果正则表达式成功匹配了一个空字符串，也会返回""。如果需要区分这种情况，请使用FindStringIndex 或FindStringSubmatch。

Example

```
re := regexp.MustCompile("fo.?")
fmt.Printf("%q\n", re.FindString("seafood"))
fmt.Printf("%q\n", re.FindString("meat"))
```

Output:

```
"foo"
""
```

func (*Regexp) FindIndex

```
func (re *Regexp) FindIndex(b []byte) (loc []int)
```

Find返回保管正则表达式re在b中的最左侧的一个匹配结果的起止位置的切片（显然len(loc)==2）。匹配结果可以通过起止位置对b做切片操作得到：b[loc[0]:loc[1]]。如果没有匹配到，会返回nil。

func (*Regexp) FindStringIndex

```
func (re *Regexp) FindStringIndex(s string) (loc []int)
```

Find返回保管正则表达式re在b中的最左侧的一个匹配结果的起止位置的切片（显然len(loc)==2）。匹配结果可以通过起止位置对b做切片操作得到：b[loc[0]:loc[1]]。如果没有匹配到，会返回nil。

Example

```
re := regexp.MustCompile("ab?")
fmt.Println(re.FindStringIndex("tablett"))
fmt.Println(re.FindStringIndex("foo") == nil)
```

Output:

```
[1 3]
true
```

func (*Regexp) FindReaderIndex

```
func (re *Regexp) FindReaderIndex(r io.RuneReader) (loc []int)
```

Find返回保管正则表达式re在b中的最左侧的一个匹配结果的起止位置的切片（显然len(loc)==2）。匹配结果可以在输入流r的字节偏移量loc[0]到loc[1]-1（包括二者）位置找到。如果没有匹配到，会返回nil。

func (*Regexp) FindSubmatch

```
func (re *Regexp) FindSubmatch(b []byte) [][]byte
```

Find返回一个保管正则表达式re在b中的最左侧的一个匹配结果以及（可能有的）分组匹配的结果的[][]byte切片。如果没有匹配到，会返回nil。

func (*Regexp) FindStringSubmatch

```
func (re *Regexp) FindStringSubmatch(s string) []string
```

Find返回一个保管正则表达式re在b中的最左侧的一个匹配结果以及（可能有的）分组匹配的结果的[]string切片。如果没有匹配到，会返回nil。

Example

```
re := regexp.MustCompile("a(x*)b(y|z)c")
fmt.Printf("%q\n", re.FindStringSubmatch("-axxxbyc-"))
fmt.Printf("%q\n", re.FindStringSubmatch("-abzc-"))
```

Output:

```
["axxxbyc" "xxx" "y"]
["abzc" "" "z"]
```

func (*Regexp) FindSubmatchIndex

```
func (re *Regexp) FindSubmatchIndex(b []byte) []int
```

Find返回一个保管正则表达式re在b中的最左侧的一个匹配结果以及（可能有的）分组匹配的结果的起止位置的切片。匹配结果和分组匹配结果可以通过起止位置对b做切片操作得到：b[loc[2*n]:loc[2*n+1]]。如果没有匹配到，会返回nil。

func (*Regexp) FindStringSubmatchIndex

```
func (re *Regexp) FindStringSubmatchIndex(s string) []int
```

Find返回一个保管正则表达式re在b中的最左侧的一个匹配结果以及（可能有的）分组匹配的结果的起止位置的切片。匹配结果和分组匹配结果可以通过起止位置对b做切片操作得到：b[loc[2*n]:loc[2*n+1]]。如果没有匹配到，会返回nil。

func (*Regexp) FindReaderSubmatchIndex

```
func (re *Regexp) FindReaderSubmatchIndex(r io.RuneReader) []int
```

Find返回一个保管正则表达式re在b中的最左侧的一个匹配结果以及（可能有的）分组匹配的结果的起止位置的切片。匹配结果和分组匹配结果可以在输入流r的字节偏移量loc[0]到loc[1]-1（包括二者）位置找到。如果没有匹配到，会返回nil。

func (*Regexp) FindAll

```
func (re *Regexp) FindAll(b []byte, n int) [][]byte
```

Find返回保管正则表达式re在b中的所有不重叠的匹配结果的[][]byte切片。如果没有匹配到，会返回nil。

func (*Regexp) FindAllString

```
func (re *Regexp) FindAllString(s string, n int) []string
```

Find返回保管正则表达式re在b中的所有不重叠的匹配结果的[]string切片。如果没有匹配到，会返回nil。

Example

```
re := regexp.MustCompile("a.")
fmt.Println(re.FindAllString("paranormal", -1))
fmt.Println(re.FindAllString("paranormal", 2))
fmt.Println(re.FindAllString("graal", -1))
fmt.Println(re.FindAllString("none", -1))
```

Output:

```
[ar an al]
[ar an]
[aa]
[]
```

func (*Regexp) FindAllIndex

```
func (re *Regexp) FindAllIndex(b []byte, n int) [][]int
```

Find返回保管正则表达式re在b中的所有不重叠的匹配结果的起止位置的切片。如果没有匹配到，会返回nil。

func (*Regexp) FindAllStringIndex

```
func (re *Regexp) FindAllStringIndex(s string, n int) [][]int
```

Find返回保管正则表达式re在b中的所有不重叠的匹配结果的起止位置的切片。如果没有匹配到，会返回nil。

func (*Regexp) FindAllSubmatch

```
func (re *Regexp) FindAllSubmatch(b []byte, n int) [][][]byte
```

Find返回一个保管正则表达式re在b中的所有不重叠的匹配结果及其对应的（可能有的）分组匹配的结果的[][][]byte切片。如果没有匹配到，会返回nil。

func (*Regexp) FindAllStringSubmatch

```
func (re *Regexp) FindAllStringSubmatch(s string, n int) [][]string
```

Find返回一个保管正则表达式re在b中的所有不重叠的匹配结果及其对应的（可能有的）分组匹配的结果的[][]string切片。如果没有匹配到，会返回nil。

Example

```
re := regexp.MustCompile("a(x*)b")
fmt.Printf("%q\n", re.FindAllStringSubmatch("-ab-", -1))
fmt.Printf("%q\n", re.FindAllStringSubmatch("-axxb-", -1))
fmt.Printf("%q\n", re.FindAllStringSubmatch("-ab-axb-", -1))
fmt.Printf("%q\n", re.FindAllStringSubmatch("-axxb-ab-", -1))
```

Output:

```
[["ab" ""]]
[["axxb" "xx"]]
[["ab" ""] ["axb" "x"]]
[["axxb" "xx"] ["ab" ""]]
```

func (*Regexp) FindAllSubmatchIndex

```
func (re *Regexp) FindAllSubmatchIndex(b []byte, n int) [][]int
```

Find返回一个保管正则表达式re在b中的所有不重叠的匹配结果及其对应的（可能有的）分组匹配的结果的起止位置的切片（第一层表示第几个匹配结果，完整匹配和分组匹配的起止位置对在第二层）。如果没有匹配到，会返回nil。

func (*Regexp) FindAllStringSubmatchIndex

```
func (re *Regexp) FindAllStringSubmatchIndex(s string, n int) [][]i
```

Find返回一个保管正则表达式re在b中的所有不重叠的匹配结果及其对应的（可能有的）分组匹配的结果的起止位置的切片（第一层表示第几个匹配结果，完整匹配和分组匹配的起止位置对在第二层）。如果没有匹配到，会返回nil。

Example

```
re := regexp.MustCompile("a(x*)b")
// Indices:
//    01234567   012345678
//    -ab-axb-   -axxb-ab-
fmt.Println(re.FindAllStringSubmatchIndex("-ab-", -1))
fmt.Println(re.FindAllStringSubmatchIndex("-axxb-", -1))
fmt.Println(re.FindAllStringSubmatchIndex("-ab-axb-", -1))
fmt.Println(re.FindAllStringSubmatchIndex("-axxb-ab-", -1))
fmt.Println(re.FindAllStringSubmatchIndex("-foo-", -1))
```

Output:

```
[[1 3 2 2]]
[[1 5 2 4]]
[[1 3 2 2] [4 7 5 6]]
[[1 5 2 4] [6 8 7 7]]
[]
```

func (*Regexp) Split

```
func (re *Regexp) Split(s string, n int) []string
```

Split将re在s中匹配到的结果作为分隔符将s分割成多个字符串，并返回这些正则匹配结果之间的字符串的切片。

返回的切片不会包含正则匹配的结果，只包含匹配结果之间的片段。当正则表达式re中不含正则元字符时，本方法等价于strings.SplitN。

举例：

```
s := regexp.MustCompile("a*").Split("abaabaccadaaae", 5)
// s: ["", "b", "b", "c", "cadaaae"]
```

参数n绝对返回的子字符串的数量：

```
n > 0 : 返回最多n个子字符串，最后一个子字符串是剩余未进行分割的部分。
n == 0: 返回nil (zero substrings)
n < 0 : 返回所有子字符串
```

func (*Regexp) Expand

```
func (re *Regexp) Expand(dst []byte, template []byte, src []byte, r
```

Expand返回新生成的将template添加到dst后面的切片。在添加时，Expand会将template中的变量替换为从src匹配的结果。match应该是被FindSubmatchIndex返回的匹配结果起止位置索引。（通常就是匹配src，除非你要将匹配得到的位置用于另一个[]byte）

在template参数里，一个变量表示为格式如：$name或${name}的字符串，其中name是长度>0的字母、数字和下划线的序列。一个单纯的数字字符名如$1会作为捕获分组的数字索引；其他的名字对应(?P<name>...)语法产生的命名捕获分组的名字。超出范围的数字索引、索引对应的分组未匹配到文本、正则表达式中未出现的分组名，都会被替换为空切片。

$name格式的变量名，name会尽可能取最长序列：$1x等价于${1x}而非${1}x，$10等价于${10}而非${1}0。因此$name适用在后跟空格/换行等字符的情况，${name}适用所有情况。

如果要在输出中插入一个字面值'$'，在template里可以使用$$。

func (*Regexp) ExpandString

```
func (re *Regexp) ExpandString(dst []byte, template string, src str
```

ExpandString类似Expand，但template和src参数为字符串。它将替换结果添加到切片并返回切片，以便让调用代码控制内存申请。

func (*Regexp) ReplaceAllLiteral

```
func (re *Regexp) ReplaceAllLiteral(src, repl []byte) []byte
```

ReplaceAllLiteral返回src的一个拷贝，将src中所有re的匹配结果都替换为repl。repl参数被直接使用，不会使用Expand进行扩展。

func (*Regexp) ReplaceAllLiteralString

```
func (re *Regexp) ReplaceAllLiteralString(src, repl string) string
```

ReplaceAllLiteralString返回src的一个拷贝，将src中所有re的匹配结果都替换为repl。repl参数被直接使用，不会使用Expand进行扩展。

Example

```
re := regexp.MustCompile("a(x*)b")
fmt.Println(re.ReplaceAllLiteralString("-ab-axxb-", "T"))
fmt.Println(re.ReplaceAllLiteralString("-ab-axxb-", "$1"))
fmt.Println(re.ReplaceAllLiteralString("-ab-axxb-", "${1}"))
```

Output:

```
-T-T-
-$1-$1-
-${1}-${1}-
```

func (*Regexp) ReplaceAll

```
func (re *Regexp) ReplaceAll(src, repl []byte) []byte
```

ReplaceAllLiteral返回src的一个拷贝，将src中所有re的匹配结果都替换为repl。在替换时，repl中的'$'符号会按照Expand方法的规则进行解释和替换，例如$1会被替换为第一个分组匹配结果。

func (*Regexp) ReplaceAllString

```
func (re *Regexp) ReplaceAllString(src, repl string) string
```

ReplaceAllLiteral返回src的一个拷贝，将src中所有re的匹配结果都替换为repl。在替换时，repl中的'$'符号会按照Expand方法的规则进行解释和替换，例如$1会被替换为第一个分组匹配结果。

Example

```
re := regexp.MustCompile("a(x*)b")
fmt.Println(re.ReplaceAllString("-ab-axxb-", "T"))
fmt.Println(re.ReplaceAllString("-ab-axxb-", "$1"))
fmt.Println(re.ReplaceAllString("-ab-axxb-", "$1W"))
fmt.Println(re.ReplaceAllString("-ab-axxb-", "${1}W"))
```

Output:

```
-T-T-
--xx-
---
-W-xxW-
```

func (*Regexp) ReplaceAllFunc

```
func (re *Regexp) ReplaceAllFunc(src []byte, repl func([]byte) []by
```

ReplaceAllLiteral返回src的一个拷贝，将src中所有re的匹配结果（设为matched）都替换为repl(matched)。repl返回的切片被直接使用，不会使用Expand进行扩展。

func (*Regexp) ReplaceAllStringFunc

```
func (re *Regexp) ReplaceAllStringFunc(src string, repl func(string
```

ReplaceAllLiteral返回src的一个拷贝，将src中所有re的匹配结果（设为matched）都替换为repl(matched)。repl返回的字符串被直接使用，不会使用Expand进行扩展。

package runtime

```
import "runtime"
```

runtime包提供和go运行时环境的互操作，如控制go程的函数。它也包括用于reflect包的低层次类型信息；参见reflect报的文档获取运行时类型系统的可编程接口。

Environment Variables

下面的环境变量（$name或%name%，这依赖于主机的操作系统）控制go程序的运行时行为。它们的含义和用法可能在各发行版之间改变。

环境变量GOGC设置最初的垃圾收集目标百分比。当新申请的数据和前次垃圾收集剩下的存活数据的比率达到该百分比时，就会触发垃圾收集。默认GOGC=100。设置GOGC=off 会完全关闭垃圾收集。runtime/debug包的SetGCPercent函数允许在运行时修改该百分比。参见http://golang.org/pkg/runtime/debug/#SetGCPercent

环境变量GODEBUG控制运行时的debug输出。GODEBUG的值是逗号分隔的name=val对。支持的name如下：

```
allocfreetrace: 设置其为1，会导致每次分配都会被记录剖面，会记录每一个对象的分
  及其堆栈踪迹。
efence: 设置其为1，会导致分配器运行模式为：每个对象申请在独立的页和地址，且永不
gctrace: 设置其为1，会导致垃圾收集器每次收集都向标准错误输出写入单行的数据，概
  内存的大小和暂停的总时间长度。设置其为2，会写入同样的概述，但也会写入每次收集
gcdead: 设置其为1，会导致垃圾收集器摧毁任何它认为已经死掉的执行堆栈。
schedtrace: 设置其为X，会导致调度程序每隔X毫秒输出单行信息到标准错误输出，概述
scheddetail: 设置schedtrace为X并设置其为1，会导致调度程序每隔X毫秒输出详细
  描述调度、进程、线程和go程的状态。
```

环境变量GOMAXPROCS限制可以同时运行用户层次的go代码的操作系统进程数。没有对代表go代码的、可以在系统调用中阻塞的go程数的限制；那些阻塞的go程不与GOMAXPROCS限制冲突。本包的GOMAXPROCS函数可以查询和修改该限制。

环境变量GOTRACEBACK控制当go程序因为不能恢复的panic或不期望的运行时情况失败时的输出。失败的程序默认会打印所有现存go程的堆栈踪迹（省略运行时系统中的函数），然后以状态码2退出。如果GOTRACEBACK为0，会完全忽略所有go程的堆栈踪迹。如果GOTRACEBACK为1，会采用默认行为。如果GOTRACEBACK为2，会打印所有现存go程包括运行时函数的堆栈踪迹。如果GOTRACEBACK为crash，会打印所有现存go程包括运行时函数的堆栈踪迹，并且如果可能会采用操作系统特定的方式崩溃，而不是退出。例如，在Unix系统里，程序会释放SIGABRT信号以触发核心信息转储。

环境变量GOARCH、GOOS、GOPATH和GOROOT构成完整的go环境变量集合。它们影响go程序的构建（参见http://golang.org/cmd/go andhttp://golang.org/pkg/go/build）。

GOARCH、GOOS和GOROOT在编译时被记录并可用本包的常量和函数获取，但它们不会影响运行时环境。

Index

Constants

```
const Compiler = "gc"
```

Compiler是编译工具链的名字，工具链会构建可执行的二进制文件。已知的工具链是：

```
gc      The 5g/6g/8g compiler suite at code.google.com/p/go.
gccgo   The gccgo front end, part of the GCC compiler suite.
```

```
const GOARCH string = theGoarch
```

GOARCH是可执行程序的目标处理器架构（将要在该架构的机器上执行）：386、amd64或arm。

```
const GOOS string = theGoos
```

GOOS是可执行程序的目标操作系统（将要在该操作系统的机器上执行）：darwin、freebsd、linux等。

Variables

```
var MemProfileRate int = 512 * 1024
```

MemProfileRate控制会在内存profile里记录和报告的内存分配采样频率。内存profile记录器平均每分配MemProfileRate字节进行一次分配采样。

要在profile里包含每一个申请的块，可以将MemProfileRate设为1。要完全关闭profile的记录，设置本变量为0。

处理内存profile的工具假设profile记录速度在整个程序的生命期是固定的，并等于当前值。修改内存profile的程序应该只进行一次，且尽可能早的修改（例如，在main函数的开始处）。

type Error

```
type Error interface {
    error
    // RuntimeError是一个无操作的函数，仅用于区别运行时错误和普通错误。
    // 具有RuntimeError方法的错误类型就是运行时错误类型。
    RuntimeError()
}
```

Error接口用来识别运行时错误。

type TypeAssertionError

```
type TypeAssertionError struct {
    // 内含隐藏或非导出字段
}
```

TypeAssertionError表示一次失败的类型断言。

func (*TypeAssertionError) Error

```
func (e *TypeAssertionError) Error() string
```

func (*TypeAssertionError) RuntimeError

```
func (*TypeAssertionError) RuntimeError()
```

func GOROOT

```
func GOROOT() string
```

GOROOT返回Go的根目录。如果存在GOROOT环境变量，返回该变量的值；否则，返回创建Go时的根目录。

func Version

```
func Version() string
```

返回Go的版本字符串。它要么是递交的hash和创建时的日期；要么是发行标签如"go1.3"。

func NumCPU

```
func NumCPU() int
```

NumCPU返回本地机器的逻辑CPU个数。

func GOMAXPROCS

```
func GOMAXPROCS(n int) int
```

GOMAXPROCS设置可同时执行的最大CPU数，并返回先前的设置。 若 n < 1，它就不会更改当前设置。本地机器的逻辑CPU数可通过 NumCPU 查询。本函数在调度程序优化后会去掉。

func SetCPUProfileRate

```
func SetCPUProfileRate(hz int)
```

SetCPUProfileRate设置CPU profile记录的速率为平均每秒hz次。如果hz<=0，SetCPUProfileRate会关闭profile的记录。如果记录器在执行，该速率必须在关闭之后才能修改。

绝大多数使用者应使用runtime/pprof包或testing包的-test.cpuprofile选项而非直接使用SetCPUProfileRate。

func CPUProfile

```
func CPUProfile() []byte
```

CPUProfile返回二进制CPU profile堆栈跟踪数据的下一个chunk，函数会阻塞直到该数据可用。 如果profile的记录被关闭，且在记录器开着的时候积累的profile数据都被返回了，CPUProfile会返回nil。调用者在再次调用本函数之前应先保存返回的

数据。

绝大多数使用者应使用 runtime/pprof包或testing包的-test.cpuprofile选项而非直接使用 CPUProfile。

func GC

```
func GC()
```

GC执行一次垃圾回收。

func SetFinalizer

```
func SetFinalizer(x, f interface{})
```

SetFinalizer将x的终止器设置为f。当垃圾收集器发现一个不能接触的（即引用计数为零，程序中不能再直接或间接访问该对象）具有终止器的块时，它会清理该关联（对象到终止器）并在独立go程调用f(x)。这使x再次可以接触，但没有了绑定的终止器。如果SetFinalizer没有被再次调用，下一次垃圾收集器将视x为不可接触的，并释放x。

SetFinalizer(x, nil)会清理任何绑定到x的终止器。

参数x必须是一个指向通过new申请的对象的指针，或者通过对复合字面值取址得到的指针。参数f必须是一个函数，它接受单个可以直接用x类型值赋值的参数，也可以有任意个被忽略的返回值。如果这两条任一条不被满足，本函数就会中断程序。

终止器会按依赖顺序执行：如果A指向B，两者都有终止器，且它们无法从其它方面接触，只有A的终止器执行；A被释放后，B的终止器就可以执行。如果一个循环结构包含一个具有终止器的块，该循环不能保证会被当垃圾收集，终止器也不能保证会执行；因为没有尊重依赖关系的顺序。

x的终止器会在x变为不可接触之后的任意时间被调度执行。不保证终止器会在程序退出前执行，因此一般终止器只用于在长期运行的程序中释放关联到某对象的非内存资源。例如，当一个程序丢弃一个os.File对象时没有调用其Close方法，该os.File对象可以使用终止器去关闭对应的操作系统文件描述符。但依靠终止器去刷新内存中的I/O缓冲如bufio.Writer是错误的，因为缓冲不会在程序退出时被刷新。

如果*x的大小为0字节，不保证终止器会执行。

一个程序会有单独一个go程顺序执行所有的终止器。如果一个终止器必须运行较长时间，它应该在内部另开go程执行该任务。

type MemStats

```
type MemStats struct {
    // 一般统计
    Alloc      uint64 // 已申请且仍在使用的字节数
    TotalAlloc uint64 // 已申请的总字节数（已释放的部分也算在内）
    Sys        uint64 // 从系统中获取的字节数（下面XxxSys之和）
    Lookups    uint64 // 指针查找的次数
    Mallocs    uint64 // 申请内存的次数
    Frees      uint64 // 释放内存的次数
    // 主分配堆统计
    HeapAlloc    uint64 // 已申请且仍在使用的字节数
    HeapSys      uint64 // 从系统中获取的字节数
    HeapIdle     uint64 // 闲置span中的字节数
    HeapInuse    uint64 // 非闲置span中的字节数
    HeapReleased uint64 // 释放到系统的字节数
    HeapObjects  uint64 // 已分配对象的总个数
    // L低层次、大小固定的结构体分配器统计，Inuse为正在使用的字节数，Sys为从系
    StackInuse  uint64 // 引导程序的堆栈
    StackSys    uint64
    MSpanInuse  uint64 // mspan结构体
    MSpanSys    uint64
    MCacheInuse uint64 // mcache结构体
    MCacheSys   uint64
    BuckHashSys uint64 // profile桶散列表
    GCSys       uint64 // GC元数据
    OtherSys    uint64 // 其他系统申请
    // 垃圾收集器统计
    NextGC       uint64 // 会在HeapAlloc字段到达该值（字节数）时运行下次G
    LastGC       uint64 // 上次运行的绝对时间（纳秒）
    PauseTotalNs uint64
    PauseNs      [256]uint64 // 近期GC暂停时间的循环缓冲，最近一次在[(Nu
    NumGC        uint32
    EnableGC     bool
    DebugGC      bool
    // 每次申请的字节数的统计，61是C代码中的尺寸分级数
    BySize [61]struct {
        Size    uint32
        Mallocs uint64
        Frees   uint64
    }
}
```

MemStats记录内存申请和分配的统计信息。

func ReadMemStats

```
func ReadMemStats(m *MemStats)
```

ReadMemStats将内存申请和分配的统计信息填写进m。

type MemProfileRecord

```
type MemProfileRecord struct {
    AllocBytes, FreeBytes     int64       // 申请和释放的字节数
    AllocObjects, FreeObjects int64       // 申请和释放的对象数
    Stack0                    [32]uintptr // 该记录的调用栈踪迹，以第一
}
```

MemProfileRecord用于描述某个调用栈序列申请和释放的活动对象等信息。

func (*MemProfileRecord) InUseBytes

```
func (r *MemProfileRecord) InUseBytes() int64
```

InUseBytes返回正在使用的字节数（AllocBytes – FreeBytes）

func (*MemProfileRecord) InUseObjects

```
func (r *MemProfileRecord) InUseObjects() int64
```

InUseObjects返回正在使用的对象数（AllocObjects - FreeObjects）

func (*MemProfileRecord) Stack

```
func (r *MemProfileRecord) Stack() []uintptr
```

Stack返回关联至此记录的调用栈踪迹，即r.Stack0的前缀。

func MemProfile

```
func MemProfile(p []MemProfileRecord, inuseZero bool) (n int, ok bo
```

MemProfile返回当前内存profile中的记录数n。若len(p)>=n，MemProfile会将此分析报告复制到p中并返回(n, true)；如果len(p)<n，MemProfile则不会更改p，而只返回(n, false)。

如果inuseZero为真，该profile就会包含无效分配记录（其中r.AllocBytes>0，而r.AllocBytes==r.FreeBytes。这些内存都是被申请后又释放回运行时环境的）。

大多数调用者应当使用runtime/pprof包或testing包的-test.memprofile标记，而非直接调用MemProfile。

func Breakpoint

```
func Breakpoint()
```

Breakpoint执行一个断点陷阱。

func Stack

```
func Stack(buf []byte, all bool) int
```

Stack将调用其的go程的调用栈踪迹格式化后写入到buf中并返回写入的字节数。若all为true，函数会在写入当前go程的踪迹信息后，将其它所有go程的调用栈踪迹都格式化写入到buf中。

func Caller

```
func Caller(skip int) (pc uintptr, file string, line int, ok bool)
```

Caller报告当前go程调用栈所执行的函数的文件和行号信息。实参skip为上溯的栈帧数，0表示Caller的调用者（Caller所在的调用栈）。（由于历史原因，skip的意思在Caller和Callers中并不相同。）函数的返回值为调用栈标识符、文件名、该调用在文件中的行号。如果无法获得信息，ok会被设为false。

func Callers

```
func Callers(skip int, pc []uintptr) int
```

函数把当前go程调用栈上的调用栈标识符填入切片pc中，返回写入到pc中的项数。实参skip为开始在pc中记录之前所要跳过的栈帧数，0表示Callers自身的调用栈，1表示Callers所在的调用栈。返回写入p的项数。

type StackRecord

```
type StackRecord struct {
    Stack0 [32]uintptr // 该记录的调用栈踪迹，以第一个零值成员截止
}
```

StackRecord描述单条调用栈。

func (*StackRecord) Stack

```
func (r *StackRecord) Stack() []uintptr
```

Stack返回与记录相关联的调用栈踪迹，即r.Stack0的前缀。

type Func

```
type Func struct {
    // 内含隐藏或非导出字段
}
```

func FuncForPC

```
func FuncForPC(pc uintptr) *Func
```

FuncForPC返回一个表示调用栈标识符pc对应的调用栈的*Func；如果该调用栈标识符没有对应的调用栈，函数会返回nil。每一个调用栈必然是对某个函数的调用。

func (*Func) Name

```
func (f *Func) Name() string
```

Name返回该调用栈所调用的函数的名字。

func (*Func) FileLine

```
func (f *Func) FileLine(pc uintptr) (file string, line int)
```

FileLine返回该调用栈所调用的函数的源代码文件名和行号。如果pc不是f内的调用栈标识符，结果是不精确的。

func (*Func) Entry

```
func (f *Func) Entry() uintptr
```

Entry返回该调用栈的调用栈标识符。

func NumCgoCall

```
func NumCgoCall() int64
```

NumCgoCall返回当前进程执行的cgo调用次数。

func NumGoroutine

```
func NumGoroutine() int
```

NumGoroutine返回当前存在的Go程数。

func Goexit

```
func Goexit()
```

Goexit终止调用它的go程。其它go程不会受影响。Goexit会在终止该go程前执行所有defer的函数。

在程序的main go程调用本函数，会终结该go程，而不会让main返回。因为main函数没有返回，程序会继续执行其它的go程。如果所有其它go程都退出了，程序就会崩溃。

func Gosched

```
func Gosched()
```

Gosched使当前go程放弃处理器，以让其它go程运行。它不会挂起当前go程，因此当前go程未来会恢复执行。

func GoroutineProfile

```
func GoroutineProfile(p []StackRecord) (n int, ok bool)
```

GoroutineProfile返回活跃go程的堆栈profile中的记录个数。若len(p) >= n，函数就会将profile中的记录复制到p中并返回(n, true)。若len(p) < n，则不会修改p，而只返回(n, false)。

绝大多数调用者应当使用runtime/pprof包，而非直接调用GoroutineProfile。

func LockOSThread

```
func LockOSThread()
```

将调用的go程绑定到它当前所在的操作系统线程。除非调用的go程退出或调用UnlockOSThread，否则它将总是在该线程中执行，而其它go程则不能进入该线程。

func UnlockOSThread

```
func UnlockOSThread()
```

将调用的go程解除和它绑定的操作系统线程。若调用的go程未调用LockOSThread，UnlockOSThread不做操作。

func ThreadCreateProfile

```
func ThreadCreateProfile(p []StackRecord) (n int, ok bool)
```

返回线程创建profile中的记录个数。如果len(p)>=n，本函数就会将profile中的记录复制到p中并返回(n, true)。若len(p)<n，则不会更改p，而只返回(n, false)。

绝大多数使用者应当使用runtime/pprof包，而非直接调用ThreadCreateProfile。

type BlockProfileRecord

```
type BlockProfileRecord struct {
    Count  int64
    Cycles int64
    StackRecord
}
```

BlockProfileRecord用于描述某个调用栈序列发生的阻塞事件的信息。

func SetBlockProfileRate

```
func SetBlockProfileRate(rate int)
```

SetBlockProfileRate控制阻塞profile记录go程阻塞事件的采样频率。对于一个阻塞事件，平均每阻塞rate纳秒，阻塞profile记录器就采集一份样本。

要在profile中包括每一个阻塞事件，需传入rate=1；要完全关闭阻塞profile的记录，需传入rate<=0。

func BlockProfile

```
func BlockProfile(p []BlockProfileRecord) (n int, ok bool)
```

BlockProfile返回当前阻塞profile中的记录个数。如果len(p)>=n，本函数就会将此profile中的记录复制到p中并返回(n, true)。如果len(p)<n，本函数则不会修改p，而只返回(n, false)。

绝大多数使用者应当使用runtime/pprof包或testing包的-test.blockprofile标记， 而非直接调用 BlockProfile。

package cgo

```
import "runtime/cgo"
```

cgo 包含有 cgo 工具生成的代码的运行时支持。 使用cgo来查看关于cgo命令的详情。

Index

package debug

```
import "runtime/debug"
```

Package debug contains facilities for programs to debug themselves while they are running.

Index

func FreeOSMemory

```
func FreeOSMemory()
```

FreeOSMemory强制进行一次垃圾收集，以释放尽量多的内存回操作系统。（即使没有调用，运行时环境也会在后台任务里逐渐将内存释放给系统）

func SetGCPercent

```
func SetGCPercent(percent int) int
```

SetGCPercent设定垃圾收集的目标百分比：当新申请的内存大小占前次垃圾收集剩余可用内存大小的比率达到设定值时，就会触发垃圾收集。SetGCPercent返回之前的设定。初始值设定为环境变量GOGC的值；如果没有设置该环境变量，初始值为100。percent参数如果是负数值，会关闭垃圾收集。

func SetMaxStack

```
func SetMaxStack(bytes int) int
```

SetMaxStack设置该以被单个go程调用栈可使用的内存最大值。如果任何go程在增加其调用栈时超出了该限制，程序就会崩溃。SetMaxStack返回之前的设置。默认设置在32位系统是250MB，在64位系统是1GB。

SetMaxStack主要用于限制无限递归的go程带来的灾难。 它只会限制未来增长的调用栈。

func SetMaxThreads

```
func SetMaxThreads(threads int) int
```

SetMaxThreads设置go程序可以使用的最大操作系统线程数。如果程序试图使用超过该限制的线程数，就会导致程序崩溃。SetMaxThreads返回之前的设置，初始设置为10000个线程。

该限制控制操作系统线程数，而非go程数。go程序只有在一个go程准备要执行，但现有的线程都阻塞在系统调用、cgo调用或被runtime.LockOSThread函数阻塞在其他go程时，才会创建一个新的线程。

SetMaxThreads主要用于限制程序无限制的创造线程导致的灾难。目的是让程序在干掉操作系统之前，先干掉它自己。

func SetPanicOnFault

```
func SetPanicOnFault(enabled bool) bool
```

SetPanicOnFault控制程序在不期望（非nil）的地址出错时的运行时行为。这些错误一般是因为运行时内存破坏的bug引起的，因此默认反应是使程序崩溃。使用内存映射的文件或进行内存的不安全操作的程序可能会在非nil的地址出现错误；SetPanicOnFault允许这些程序请求运行时只触发一个panic，而不是崩溃。SetPanicOnFault只用于当前的go程。它返回之前的设置。

type GCStats

```
type GCStats struct {
    LastGC         time.Time        // 最近一次垃圾收集的时间
    NumGC          int64            // 垃圾收集的次数
    PauseTotal     time.Duration    // 所有暂停收集垃圾消耗的总时间
    Pause          []time.Duration  // 每次暂停收集垃圾的消耗的时间
    PauseQuantiles []time.Duration
}
```

GCStats收集了近期垃圾收集的信息。

func ReadGCStats

```
func ReadGCStats(stats *GCStats)
```

ReadGCStats将垃圾收集信息填入stats里。stats.Pause字段的长度是依赖于系统的；stats.Pause切片如果长度足够会被重用，否则会重新申请。ReadGCStats可能会使用stats.Pause切片的全部容量。

如果stats.PauseQuantiles字段是非空的，ReadGCStats会在其中填写说明暂停时间分配的分位数。例如，如果len(stats.PauseQuantiles)为5，该字段会被填写上0%、25%、50%、75%、100%位置的分位数（就是说，不大于该位置暂停时间的暂停次数占总暂停次数的比例分别是0%、25%……）

func WriteHeapDump

```
func WriteHeapDump(fd uintptr)
```

WriteHeapDump将内存分配堆和其中对象的描述写入给定文件描述符fd指定的文件。

堆转储格式参见http://golang.org/s/go13heapdump

func Stack

```
func Stack() []byte
```

Stack 返回格式化的go程的调用栈踪迹。 对于每一个调用栈，它包括原文件的行信息和PC值；对go函数还会尝试获取调用该函数的函数或方法，及调用所在行的文本。

此函数已废弃。请使用runtime包中的Stack函数代替。

func PrintStack

```
func PrintStack()
```

PrintStack将Stack返回信息打印到标准错误输出。

package pprof

```
import "runtime/pprof"
```

pprof包以pprof可视化工具期望的格式书写运行时剖面数据。

pprof的更多信息参见http://code.google.com/p/google-perftools/。

Index

func Profiles

```
func Profiles() []*Profile
```

Profiles返回所有已知profile的切片，按名称排序。

func StartCPUProfile

```
func StartCPUProfile(w io.Writer) error
```

StartCPUProfile为当前进程开启CPU profile。在分析时，分析报告会缓存并写入到w中。若分析已经开启，StartCPUProfile就会返回错误。

func StopCPUProfile

```
func StopCPUProfile()
```

StopCPUProfile会停止当前的CPU profile（如果有）。StopCPUProfile 只会在所有的分析报告写入完毕后才会返回。

func WriteHeapProfile

```
func WriteHeapProfile(w io.Writer) error
```

WriteHeapProfile是Lookup("heap").WriteTo(w, 0) 的简写。它是为了保持向后兼容性而存在的。

type Profile

```
type Profile struct {
    // 内含隐藏或非导出字段
}
```

Profile是一个调用栈踪迹的集合，显示导致特定事件（如内存分配）的实例的调用栈序列。包可以创建并维护它们自己的profile；它一般用于跟踪必须被显式关闭的资源，例如文件或网络连接。

一个Profile的方法可被多个Go程同时调用。

每个Profile都有唯一的名称。有些Profile是预定义的：

```
goroutine      - 当前Go所有程的调用栈踪迹
heap           - 所有堆分配的采样
threadcreate   - 导致新的OS线程创建的调用栈踪迹
block          - 导致同步原语水平的阻塞的调用栈踪迹
```

这些预声明的Profile自我维护，如果对它们调用Add或者Remove时，将导致panic。

CPU profile不能作为Profile使用。它有专门的API，即StartCPUProfile和StopCPUProfile函数，因为它在分析时是以流的形式输出到writer中的。

func Lookup

```
func Lookup(name string) *Profile
```

Lookup返回具有指定名字的Profile；如果没有，会返回nil。

func NewProfile

```
func NewProfile(name string) *Profile
```

以给定的名称创建一个新的Profile。若拥有该名称的Profile已存在，NewProfile就会panic。

约定会使用'import/path.'前缀来为每个包创建单独的命名空间。

func (*Profile) Name

```
func (p *Profile) Name() string
```

Name返回该Profile的名称，它可被传入Lookup来重新获取该Profile。

func (*Profile) Add

```
func (p *Profile) Add(value interface{}, skip int)
```

Add 将当前的执行栈添加到该分析中，并与value关联。Add在一个内部映射中存储值，因此value必须适于用作映射键，且在对应的Remove调用之前不会被垃圾收集。

若分析的映射中已经存在value键，Add就会引发panic。

skip 参数与runtime.Caller的skip参数意义相同，它用于控制调用栈踪迹从哪里开始。skip=0时候会从调用Add的函数开始。例如，给出如下执行栈：

```
Add
called from rpc.NewClient
called from mypkg.Run
called from main.main
```

当skip=0时，调用栈踪迹从rpc.NewClient对Add的调用开始；当skp=1时，堆调用踪迹从对rpc.NewClient的调用开始。

func (*Profile) Count

```
func (p *Profile) Count() int
```

Count返回该Profile中当前执行栈的数量。

func (*Profile) Remove

```
func (p *Profile) Remove(value interface{})
```

Remove从该分析中移除与值value相关联的执行栈。若值value不在此分析中，则不做操作。

func (*Profile) WriteTo

```
func (p *Profile) WriteTo(w io.Writer, debug int) error
```

函数将pprof格式的profile快照写入w中。若一个向w的写入返回一个错误，WriteTo就会返回该错误；否则会返回nil。

debug 参数用于开启附加的输出。如果debug=0，只会打印pprof所需要的十六进制地址；如果debug=1，会将地址翻译为函数名和行号并添加注释，以便让程序员无需工具阅读分析报告。

预定义Profile为其它debug值赋予了含义。例如，当打印“Go程”的分析报告时，debug=2意为：由于不可恢复的恐慌而濒临崩溃时，使用与Go程序相同的格式打印Go程的堆栈信息。

Bugs

☞ NetBSD和OS X上的profile记录服务是不完整、不准确的，参见http://golang.org/issue/6047获取细节。

package race

```
import "runtime/race"
```

race包实现了数据竞争检测逻辑。没有提供公共接口。关于race detector的详情参见 http://golang.org/doc/articles/race_detector.html

Index

package sort

```
import "sort"
```

sort包提供了排序切片和用户自定义数据集的函数。

Example

```
package sort_test
import (
    "fmt"
    "sort"
)
type Person struct {
    Name string
    Age  int
}
func (p Person) String() string {
    return fmt.Sprintf("%s: %d", p.Name, p.Age)
}
// ByAge implements sort.Interface for []Person based on
// the Age field.
type ByAge []Person
func (a ByAge) Len() int           { return len(a) }
func (a ByAge) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
func (a ByAge) Less(i, j int) bool { return a[i].Age < a[j].Age }
func Example() {
    people := []Person{
        {"Bob", 31},
        {"John", 42},
        {"Michael", 17},
        {"Jenny", 26},
    }
    fmt.Println(people)
    sort.Sort(ByAge(people))
    fmt.Println(people)
    // Output:
    // [Bob: 31 John: 42 Michael: 17 Jenny: 26]
    // [Michael: 17 Jenny: 26 Bob: 31 John: 42]
}
```

Example (SortKeys)

```
package sort_test
import (
    "fmt"
    "sort"
```

```
)
// A couple of type definitions to make the units clear.
type earthMass float64
type au float64
// A Planet defines the properties of a solar system object.
type Planet struct {
    name     string
    mass     earthMass
    distance au
}
// By is the type of a "less" function that defines the ordering of
type By func(p1, p2 *Planet) bool
// Sort is a method on the function type, By, that sorts the argume
func (by By) Sort(planets []Planet) {
    ps := &planetSorter{
        planets: planets,
        by:      by, // The Sort method's receiver is the function
    }
    sort.Sort(ps)
}
// planetSorter joins a By function and a slice of Planets to be so
type planetSorter struct {
    planets []Planet
    by      func(p1, p2 *Planet) bool // Closure used in the Less m
}
// Len is part of sort.Interface.
func (s *planetSorter) Len() int {
    return len(s.planets)
}
// Swap is part of sort.Interface.
func (s *planetSorter) Swap(i, j int) {
    s.planets[i], s.planets[j] = s.planets[j], s.planets[i]
}
// Less is part of sort.Interface. It is implemented by calling the
func (s *planetSorter) Less(i, j int) bool {
    return s.by(&s.planets[i], &s.planets[j])
}
var planets = []Planet{
    {"Mercury", 0.055, 0.4},
    {"Venus", 0.815, 0.7},
    {"Earth", 1.0, 1.0},
    {"Mars", 0.107, 1.5},
}
// ExampleSortKeys demonstrates a technique for sorting a struct ty
func Example_sortKeys() {
    // Closures that order the Planet structure.
    name := func(p1, p2 *Planet) bool {
        return p1.name < p2.name
    }
    mass := func(p1, p2 *Planet) bool {
        return p1.mass < p2.mass
    }
    distance := func(p1, p2 *Planet) bool {
```

```
        return p1.distance < p2.distance
    }
    decreasingDistance := func(p1, p2 *Planet) bool {
        return !distance(p1, p2)
    }
    // Sort the planets by the various criteria.
    By(name).Sort(planets)
    fmt.Println("By name:", planets)
    By(mass).Sort(planets)
    fmt.Println("By mass:", planets)
    By(distance).Sort(planets)
    fmt.Println("By distance:", planets)
    By(decreasingDistance).Sort(planets)
    fmt.Println("By decreasing distance:", planets)
    // Output: By name: [{Earth 1 1} {Mars 0.107 1.5} {Mercury 0.05
    // By mass: [{Mercury 0.055 0.4} {Mars 0.107 1.5} {Venus 0.815
    // By distance: [{Mercury 0.055 0.4} {Venus 0.815 0.7} {Earth 1
    // By decreasing distance: [{Mars 0.107 1.5} {Earth 1 1} {Venus
}
```

Example (SortMultiKeys)

```
package sort_test
import (
    "fmt"
    "sort"
)
// A Change is a record of source code changes, recording user, lan
type Change struct {
    user     string
    language string
    lines    int
}
type lessFunc func(p1, p2 *Change) bool
// multiSorter implements the Sort interface, sorting the changes w
type multiSorter struct {
    changes []Change
    less    []lessFunc
}
// Sort sorts the argument slice according to the less functions pa
func (ms *multiSorter) Sort(changes []Change) {
    ms.changes = changes
    sort.Sort(ms)
}
// OrderedBy returns a Sorter that sorts using the less functions,
// Call its Sort method to sort the data.
func OrderedBy(less ...lessFunc) *multiSorter {
    return &multiSorter{
        less: less,
    }
```

```
}
// Len is part of sort.Interface.
func (ms *multiSorter) Len() int {
    return len(ms.changes)
}
// Swap is part of sort.Interface.
func (ms *multiSorter) Swap(i, j int) {
    ms.changes[i], ms.changes[j] = ms.changes[j], ms.changes[i]
}
// Less is part of sort.Interface. It is implemented by looping alo
// less functions until it finds a comparison that is either Less o
// !Less. Note that it can call the less functions twice per call.
// could change the functions to return -1, 0, 1 and reduce the
// number of calls for greater efficiency: an exercise for the reac
func (ms *multiSorter) Less(i, j int) bool {
    p, q := &ms.changes[i], &ms.changes[j]
    // Try all but the last comparison.
    var k int
    for k = 0; k < len(ms.less)-1; k++ {
        less := ms.less[k]
        switch {
        case less(p, q):
            // p < q, so we have a decision.
            return true
        case less(q, p):
            // p > q, so we have a decision.
            return false
        }
        // p == q; try the next comparison.
    }
    // All comparisons to here said "equal", so just return whateve
    // the final comparison reports.
    return ms.less[k](p, q)
}
var changes = []Change{
    {"gri", "Go", 100},
    {"ken", "C", 150},
    {"glenda", "Go", 200},
    {"rsc", "Go", 200},
    {"r", "Go", 100},
    {"ken", "Go", 200},
    {"dmr", "C", 100},
    {"r", "C", 150},
    {"gri", "Smalltalk", 80},
}
// ExampleMultiKeys demonstrates a technique for sorting a struct t
// sets of multiple fields in the comparison. We chain together "Le
// which compares a single field.
func Example_sortMultiKeys() {
    // Closures that order the Change structure.
    user := func(c1, c2 *Change) bool {
        return c1.user < c2.user
    }
```

```
    language := func(c1, c2 *Change) bool {
        return c1.language < c2.language
    }
    increasingLines := func(c1, c2 *Change) bool {
        return c1.lines < c2.lines
    }
    decreasingLines := func(c1, c2 *Change) bool {
        return c1.lines > c2.lines // Note: > orders downwards.
    }
    // Simple use: Sort by user.
    OrderedBy(user).Sort(changes)
    fmt.Println("By user:", changes)
    // More examples.
    OrderedBy(user, increasingLines).Sort(changes)
    fmt.Println("By user,<lines:", changes)
    OrderedBy(user, decreasingLines).Sort(changes)
    fmt.Println("By user,>lines:", changes)
    OrderedBy(language, increasingLines).Sort(changes)
    fmt.Println("By language,<lines:", changes)
    OrderedBy(language, increasingLines, user).Sort(changes)
    fmt.Println("By language,<lines,user:", changes)
    // Output:
    // By user: [{dmr C 100} {glenda Go 200} {gri Smalltalk 80} {gr
    // By user,<lines: [{dmr C 100} {glenda Go 200} {gri Smalltalk
    // By user,>lines: [{dmr C 100} {glenda Go 200} {gri Go 100} {g
    // By language,<lines: [{dmr C 100} {ken C 150} {r C 150} {gri
    // By language,<lines,user: [{dmr C 100} {ken C 150} {r C 150}
}
```

Example (SortWrapper)

```
package sort_test
import (
    "fmt"
    "sort"
)
type Grams int
func (g Grams) String() string { return fmt.Sprintf("%dg", int(g))
type Organ struct {
    Name   string
    Weight Grams
}
type Organs []*Organ
func (s Organs) Len() int      { return len(s) }
func (s Organs) Swap(i, j int) { s[i], s[j] = s[j], s[i] }
// ByName implements sort.Interface by providing Less and using the
// Swap methods of the embedded Organs value.
type ByName struct{ Organs }
func (s ByName) Less(i, j int) bool { return s.Organs[i].Name < s.O
// ByWeight implements sort.Interface by providing Less and using t
```

```
// Swap methods of the embedded Organs value.
type ByWeight struct{ Organs }
func (s ByWeight) Less(i, j int) bool { return s.Organs[i].Weight <
func Example_sortWrapper() {
    s := []*Organ{
        {"brain", 1340},
        {"heart", 290},
        {"liver", 1494},
        {"pancreas", 131},
        {"prostate", 62},
        {"spleen", 162},
    }
    sort.Sort(ByWeight{s})
    fmt.Println("Organs by weight:")
    printOrgans(s)
    sort.Sort(ByName{s})
    fmt.Println("Organs by name:")
    printOrgans(s)
    // Output:
    // Organs by weight:
    // prostate (62g)
    // pancreas (131g)
    // spleen   (162g)
    // heart    (290g)
    // brain    (1340g)
    // liver    (1494g)
    // Organs by name:
    // brain    (1340g)
    // heart    (290g)
    // liver    (1494g)
    // pancreas (131g)
    // prostate (62g)
    // spleen   (162g)
}
func printOrgans(s []*Organ) {
    for _, o := range s {
        fmt.Printf("%-8s (%v)\n", o.Name, o.Weight)
    }
}
```

Index

Examples

type Interface

```
type Interface interface {
    // Len方法返回集合中的元素个数
    Len() int
    // Less方法报告索引i的元素是否比索引j的元素小
    Less(i, j int) bool
    // Swap方法交换索引i和j的两个元素
    Swap(i, j int)
}
```

一个满足sort.Interface接口的（集合）类型可以被本包的函数进行排序。方法要求集合中的元素可以被整数索引。

type IntSlice

```
type IntSlice []int
```

IntSlice给[]int添加方法以满足Interface接口，以便排序为递增序列。

func (IntSlice) Len

```
func (p IntSlice) Len() int
```

func (IntSlice) Less

```
func (p IntSlice) Less(i, j int) bool
```

func (IntSlice) Swap

```
func (p IntSlice) Swap(i, j int)
```

func (IntSlice) Sort

```
func (p IntSlice) Sort()
```

Sort等价于调用Sort(p)

func (IntSlice) Search

```
func (p IntSlice) Search(x int) int
```

Search等价于调用SearchInts(p, x)

type Float64Slice

```
type Float64Slice []float64
```

Float64Slice给[]float64添加方法以满足Interface接口，以便排序为递增序列。

func (Float64Slice) Len

```
func (p Float64Slice) Len() int
```

func (Float64Slice) Less

```
func (p Float64Slice) Less(i, j int) bool
```

func (Float64Slice) Swap

```
func (p Float64Slice) Swap(i, j int)
```

func (Float64Slice) Sort

```
func (p Float64Slice) Sort()
```

Sort等价于调用Sort(p)

func (Float64Slice) Search

```
func (p Float64Slice) Search(x float64) int
```

Search等价于调用SearchFloat64s(p, x)

type StringSlice

```
type StringSlice []string
```

StringSlice给[]string添加方法以满足Interface接口，以便排序为递增序列。

func (StringSlice) Len

```
func (p StringSlice) Len() int
```

func (StringSlice) Less

```
func (p StringSlice) Less(i, j int) bool
```

func (StringSlice) Swap

```
func (p StringSlice) Swap(i, j int)
```

func (StringSlice) Sort

```
func (p StringSlice) Sort()
```

Sort等价于调用Sort(p)

func (StringSlice) Search

```
func (p StringSlice) Search(x string) int
```

Search等价于调用SearchStrings(p, x)

func Sort

```
func Sort(data Interface)
```

Sort排序data。它调用1次data.Len确定长度，调用O(n*log(n))次data.Less和data.Swap。本函数不能保证排序的稳定性（即不保证相等元素的相对次序不变）。

func Stable

```
func Stable(data Interface)
```

Stable排序data，并保证排序的稳定性，相等元素的相对次序不变。

它调用1次data.Len，O(n*log(n))次data.Less和O(n*log(n)*log(n))次data.Swap。

func IsSorted

```
func IsSorted(data Interface) bool
```

IsSorted报告data是否已经被排序。

func Reverse

```
func Reverse(data Interface) Interface
```

Reverse包装一个Interface接口并返回一个新的Interface接口，对该接口排序可生成递减序列。

Example

```
s := []int{5, 2, 6, 3, 1, 4} // unsorted
sort.Sort(sort.Reverse(sort.IntSlice(s)))
fmt.Println(s)
```

Output:

```
[6 5 4 3 2 1]
```

func Search

```
func Search(n int, f func(int) bool) int
```

Search函数采用二分法搜索找到[0, n)区间内最小的满足f(i)==true的值i。也就是说，Search函数希望f在输入位于区间[0, n)的前面某部分（可以为空）时返回假，而在输入位于剩余至结尾的部分（可以为空）时返回真；Search函数会返回满足

f(i)==true的最小值i。如果没有该值，函数会返回n。注意，未找到时的返回值不是-1，这一点和strings.Index等函数不同。Search函数只会用区间[0, n)内的值调用f。

一般使用Search找到值x在插入一个有序的、可索引的数据结构时，应插入的位置。这种情况下，参数f（通常是闭包）会捕捉应搜索的值和被查询的数据集。

例如，给定一个递增顺序的切片，调用Search(len(data), func(i int) bool { return data[i] >= 23 })会返回data中最小的索引i满足data[i] >= 23。如果调用者想要知道23是否在切片里，它必须另外检查data[i] == 23。

搜索递减顺序的数据时，应使用<=运算符代替>=运算符。

下列代码尝试在一个递增顺序的整数切片中找到值x：

```
x := 23
i := sort.Search(len(data), func(i int) bool { return data[i] >= x
if i < len(data) && data[i] == x {
    // x is present at data[i]
} else {
    // x is not present in data,
    // but i is the index where it would be inserted.
}
```

一个更古怪的例子，下面的程序会猜测你持有的数字：

```
func GuessingGame() {
    var s string
    fmt.Printf("Pick an integer from 0 to 100.\n")
    answer := sort.Search(100, func(i int) bool {
        fmt.Printf("Is your number <= %d? ", i)
        fmt.Scanf("%s", &s)
        return s != "" && s[0] == 'y'
    })
    fmt.Printf("Your number is %d.\n", answer)
}
```

func Ints

```
func Ints(a []int)
```

Ints函数将a排序为递增顺序。

Example

```
s := []int{5, 2, 6, 3, 1, 4} // unsorted
sort.Ints(s)
fmt.Println(s)
```

Output:

```
[1 2 3 4 5 6]
```

func IntsAreSorted

```
func IntsAreSorted(a []int) bool
```

IntsAreSorted检查a是否已排序为递增顺序。

func SearchInts

```
func SearchInts(a []int, x int) int
```

SearchInts在递增顺序的a中搜索x，返回x的索引。如果查找不到，返回值是x应该插入a的位置（以保证a的递增顺序），返回值可以是len(a)。

func Float64s

```
func Float64s(a []float64)
```

Float64s函数将a排序为递增顺序。

func Float64sAreSorted

```
func Float64sAreSorted(a []float64) bool
```

Float64sAreSorted检查a是否已排序为递增顺序。

func SearchFloat64s

```
func SearchFloat64s(a []float64, x float64) int
```

SearchFloat64s在递增顺序的a中搜索x，返回x的索引。如果查找不到，返回值是x应该插入a的位置（以保证a的递增顺序），返回值可以是len(a)。

func Strings

```
func Strings(a []string)
```

Strings函数将a排序为递增顺序。

func StringsAreSorted

```
func StringsAreSorted(a []string) bool
```

StringsAreSorted检查a是否已排序为递增顺序。

func SearchStrings

```
func SearchStrings(a []string, x string) int
```

SearchStrings在递增顺序的a中搜索x，返回x的索引。如果查找不到，返回值是x应该插入a的位置（以保证a的递增顺序），返回值可以是len(a)。

package strconv

```
import "strconv"
```

strconv包实现了基本数据类型和其字符串表示的相互转换。

Index

Examples

Constants

```
const IntSize = intSize
```

IntSize是int或uint类型的字位数。

Variables

```
var ErrRange = errors.New("value out of range")
```

ErrRange表示超出目标类型表示范围。

```
var ErrSyntax = errors.New("invalid syntax")
```

ErrSyntax表示不符合目标类型语法。

type NumError

```
type NumError struct {
    Func string // 失败的函数（ParseBool、ParseInt、ParseUint、ParseFl
    Num  string // 输入的字符串
    Err  error  // 失败的原因（ErrRange、ErrSyntax）
}
```

NumError表示一次失败的转换。

func (*NumError) Error

```
func (e *NumError) Error() string
```

func IsPrint

```
func IsPrint(r rune) bool
```

返回一个字符是否是可打印的，和unicode.IsPrint一样，r必须是：字母（广义）、数字、标点、符号、ASCII空格。

func CanBackquote

```
func CanBackquote(s string) bool
```

返回字符串s是否可以不被修改的表示为一个单行的、没有空格和tab之外控制字符的反引号字符串。

func Quote

```
func Quote(s string) string
```

返回字符串s在go语法下的双引号字面值表示，控制字符、不可打印字符会进行转义。（如\t，\n，\xFF，\u0100）

func QuoteToASCII

```
func QuoteToASCII(s string) string
```

返回字符串s在go语法下的双引号字面值表示，控制字符和不可打印字符、非ASCII字符会进行转义。

func QuoteRune

```
func QuoteRune(r rune) string
```

返回字符r在go语法下的单引号字面值表示，控制字符、不可打印字符会进行转义。（如\t，\n，\xFF，\u0100）

func QuoteRuneToASCII

```
func QuoteRuneToASCII(r rune) string
```

返回字符r在go语法下的单引号字面值表示，控制字符、不可打印字符、非ASCII字符会进行转义。

func Unquote

```
func Unquote(s string) (t string, err error)
```

函数假设s是一个单引号、双引号、反引号包围的go语法字符串，解析它并返回它表示的值。（如果是单引号括起来的，函数会认为s是go字符字面值，返回一个单字符的字符串）

Example

```
test := func(s string) {
    t, err := strconv.Unquote(s)
    if err != nil {
        fmt.Printf("Unquote(%#v): %v\n", s, err)
    } else {
        fmt.Printf("Unquote(%#v) = %v\n", s, t)
    }
}
s := `cafe\u0301`
// If the string doesn't have quotes, it can't be unquoted.
test(s) // invalid syntax
test("`" + s + "`")
test(`"` + s + `"`)
test(`'\u00e9'`)
```

Output:

```
Unquote("cafe\\u0301"): invalid syntax
Unquote("`cafe\\u0301`") = cafe\u0301
Unquote("\"cafe\\u0301\"") = café
Unquote("'\\u00e9'") = é
```

func UnquoteChar

```
func UnquoteChar(s string, quote byte) (value rune, multibyte bool,
```

函数假设s是一个表示字符的go语法字符串，解析它并返回四个值：

```
1) value，表示一个rune值或者一个byte值
2) multibyte，表示value是否是一个多字节的utf-8字符
3) tail，表示字符串剩余的部分
4) err，表示可能存在的语法错误
```

quote参数为单引号时，函数认为单引号是语法字符，不接受未转义的单引号；双引号时，函数认为双引号是语法字符，不接受未转义的双引号；如果是零值，函数把单引号和双引号当成普通字符。

func ParseBool

```
func ParseBool(str string) (value bool, err error)
```

返回字符串表示的bool值。它接受1、0、t、f、T、F、true、false、True、False、TRUE、FALSE；否则返回错误。

func ParseInt

```
func ParseInt(s string, base int, bitSize int) (i int64, err error)
```

返回字符串表示的整数值，接受正负号。

base指定进制（2到36），如果base为0，则会从字符串前置判断，"0x"是16进制，"0"是8进制，否则是10进制；

bitSize指定结果必须能无溢出赋值的整数类型，0、8、16、32、64 分别代表 int、int8、int16、int32、int64；返回的err是*NumErr类型的，如果语法有误，err.Error = ErrSyntax；如果结果超出类型范围err.Error = ErrRange。

func ParseUint

```
func ParseUint(s string, base int, bitSize int) (n uint64, err erro
```

ParseUint类似ParseInt但不接受正负号，用于无符号整型。

func ParseFloat

```
func ParseFloat(s string, bitSize int) (f float64, err error)
```

解析一个表示浮点数的字符串并返回其值。

如果s合乎语法规则，函数会返回最为接近s表示值的一个浮点数（使用IEEE754规范舍入）。bitSize指定了期望的接收类型，32是float32（返回值可以不改变精确值的赋值给float32），64是float64；返回值err是*NumErr类型的，语法有误的，err.Error=ErrSyntax；结果超出表示范围的，返回值f为±Inf，err.Error= ErrRange。

func FormatBool

```
func FormatBool(b bool) string
```

根据b的值返回"true"或"false"。

func FormatInt

```
func FormatInt(i int64, base int) string
```

返回i的base进制的字符串表示。base 必须在2到36之间，结果中会使用小写字母'a'到'z'表示大于10的数字。

func FormatUint

```
func FormatUint(i uint64, base int) string
```

是FormatInt的无符号整数版本。

func FormatFloat

```
func FormatFloat(f float64, fmt byte, prec, bitSize int) string
```

函数将浮点数表示为字符串并返回。

bitSize表示f的来源类型（32：float32、64：float64），会据此进行舍入。

fmt表示格式：'f'（-ddd.dddd）、'b'（-ddddp±ddd，指数为二进制）、'e'（-d.dddde±dd，十进制指数）、'E'（-d.ddddE±dd，十进制指数）、'g'（指数很大时用'e'格式，否则'f'格式）、'G'（指数很大时用'E'格式，否则'f'格式）。

prec控制精度（排除指数部分）：对'f'、'e'、'E'，它表示小数点后的数字个数；对'g'、'G'，它控制总的数字个数。如果prec 为-1，则代表使用最少数量的、但又必需的数字来表示f。

func Atoi

```
func Atoi(s string) (i int, err error)
```

Atoi是ParseInt(s, 10, 0)的简写。

func Itoa

```
func Itoa(i int) string
```

Itoa是FormatInt(i, 10) 的简写。

func AppendBool

```
func AppendBool(dst []byte, b bool) []byte
```

等价于append(dst, FormatBool(b)...)

func AppendInt

```
func AppendInt(dst []byte, i int64, base int) []byte
```

等价于append(dst, FormatInt(I, base)...)

func AppendUint

```
func AppendUint(dst []byte, i uint64, base int) []byte
```

等价于append(dst, FormatUint(I, base)...)

func AppendFloat

```
func AppendFloat(dst []byte, f float64, fmt byte, prec int, bitSize
```

等价于append(dst, FormatFloat(f, fmt, prec, bitSize)...)

func AppendQuote

```
func AppendQuote(dst []byte, s string) []byte
```

等价于append(dst, Quote(s)...)

func AppendQuoteToASCII

```
func AppendQuoteToASCII(dst []byte, s string) []byte
```

等价于append(dst, QuoteToASCII(s)...)

func AppendQuoteRune

```
func AppendQuoteRune(dst []byte, r rune) []byte
```

等价于append(dst, QuoteRune(r)...)

func AppendQuoteRuneToASCII

```
func AppendQuoteRuneToASCII(dst []byte, r rune) []byte
```

等价于append(dst, QuoteRuneToASCII(r)...)

package strings

```
import "strings"
```

strings包实现了用于操作字符的简单函数。

Index

Examples

- TrimPrefix
- TrimSpace
- TrimSuffix

func EqualFold

```
func EqualFold(s, t string) bool
```

判断两个utf-8编码字符串（将unicode大写、小写、标题三种格式字符视为相同）是否相同。

Example

```
fmt.Println(strings.EqualFold("Go", "go"))
```

Output:

```
true
```

func HasPrefix

```
func HasPrefix(s, prefix string) bool
```

判断s是否有前缀字符串prefix。

func HasSuffix

```
func HasSuffix(s, suffix string) bool
```

判断s是否有后缀字符串suffix。

func Contains

```
func Contains(s, substr string) bool
```

判断字符串s是否包含子串substr。

Example

```
fmt.Println(strings.Contains("seafood", "foo"))
fmt.Println(strings.Contains("seafood", "bar"))
fmt.Println(strings.Contains("seafood", ""))
fmt.Println(strings.Contains("", ""))
```

Output:

```
true
false
true
true
```

func ContainsRune

```
func ContainsRune(s string, r rune) bool
```

判断字符串s是否包含utf-8码值r。

func ContainsAny

```
func ContainsAny(s, chars string) bool
```

判断字符串s是否包含字符串chars中的任一字符。

Example

```
fmt.Println(strings.ContainsAny("team", "i"))
fmt.Println(strings.ContainsAny("failure", "u & i"))
fmt.Println(strings.ContainsAny("foo", ""))
fmt.Println(strings.ContainsAny("", ""))
```

Output:

```
false
true
false
false
```

func Count

```
func Count(s, sep string) int
```

返回字符串s中有几个不重复的sep子串。

Example

```
fmt.Println(strings.Count("cheese", "e"))
fmt.Println(strings.Count("five", "")) // before & after each rune
```

Output:

```
3
5
```

func Index

```
func Index(s, sep string) int
```

子串sep在字符串s中第一次出现的位置，不存在则返回-1。

Example

```
fmt.Println(strings.Index("chicken", "ken"))
fmt.Println(strings.Index("chicken", "dmr"))
```

Output:

```
4
-1
```

func IndexByte

```
func IndexByte(s string, c byte) int
```

字符c在s中第一次出现的位置，不存在则返回-1。

func IndexRune

```
func IndexRune(s string, r rune) int
```

unicode码值r在s中第一次出现的位置，不存在则返回-1。

Example

```
fmt.Println(strings.IndexRune("chicken", 'k'))
fmt.Println(strings.IndexRune("chicken", 'd'))
```

Output:

```
4
-1
```

func IndexAny

```
func IndexAny(s, chars string) int
```

字符串chars中的任一utf-8码值在s中第一次出现的位置，如果不存在或者chars为空字符串则返回-1。

Example

```
fmt.Println(strings.IndexAny("chicken", "aeiouy"))
fmt.Println(strings.IndexAny("crwth", "aeiouy"))
```

Output:

```
2
-1
```

func IndexFunc

```
func IndexFunc(s string, f func(rune) bool) int
```

s中第一个满足函数f的位置i（该处的utf-8码值r满足f(r)==true），不存在则返回-1。

Example

```
f := func(c rune) bool {
    return unicode.Is(unicode.Han, c)
}
fmt.Println(strings.IndexFunc("Hello, 世界", f))
fmt.Println(strings.IndexFunc("Hello, world", f))
```

Output:

```
7
-1
```

func LastIndex

```
func LastIndex(s, sep string) int
```

子串sep在字符串s中最后一次出现的位置，不存在则返回-1。

Example

```
fmt.Println(strings.Index("go gopher", "go"))
fmt.Println(strings.LastIndex("go gopher", "go"))
fmt.Println(strings.LastIndex("go gopher", "rodent"))
```

Output:

```
0
3
-1
```

func LastIndexAny

```
func LastIndexAny(s, chars string) int
```

字符串chars中的任一utf-8码值在s中最后一次出现的位置，如不存在或者chars为空字符串则返回-1。

func LastIndexFunc

```
func LastIndexFunc(s string, f func(rune) bool) int
```

s中最后一个满足函数f的unicode码值的位置i，不存在则返回-1。

func Title

```
func Title(s string) string
```

返回s中每个单词的首字母都改为标题格式的字符串拷贝。

BUG: Title用于划分单词的规则不能很好的处理Unicode标点符号。

Example

```
fmt.Println(strings.Title("her royal highness"))
```

Output:

```
Her Royal Highness
```

func ToLower

```
func ToLower(s string) string
```

返回将所有字母都转为对应的小写版本的拷贝。

Example

```
fmt.Println(strings.ToLower("Gopher"))
```

Output:

```
gopher
```

func ToLowerSpecial

```
func ToLowerSpecial(_case unicode.SpecialCase, s string) string
```

使用_case规定的字符映射，返回将所有字母都转为对应的小写版本的拷贝。

func ToUpper

```
func ToUpper(s string) string
```

返回将所有字母都转为对应的大写版本的拷贝。

Example

```
fmt.Println(strings.ToUpper("Gopher"))
```

Output:

```
GOPHER
```

func ToUpperSpecial

```
func ToUpperSpecial(_case unicode.SpecialCase, s string) string
```

使用_case规定的字符映射，返回将所有字母都转为对应的大写版本的拷贝。

func ToTitle

```
func ToTitle(s string) string
```

返回将所有字母都转为对应的标题版本的拷贝。

Example

```
fmt.Println(strings.ToTitle("loud noises"))
fmt.Println(strings.ToTitle("хлеб"))
```

Output:

```
LOUD NOISES
ХЛЕБ
```

func ToTitleSpecial

```
func ToTitleSpecial(_case unicode.SpecialCase, s string) string
```

使用_case规定的字符映射，返回将所有字母都转为对应的标题版本的拷贝。

func Repeat

```
func Repeat(s string, count int) string
```

返回count个s串联的字符串。

Example

```
fmt.Println("ba" + strings.Repeat("na", 2))
```

Output:

```
banana
```

func Replace

```
func Replace(s, old, new string, n int) string
```

返回将s中前n个不重叠old子串都替换为new的新字符串，如果n<0会替换所有old子串。

Example

```
fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))
fmt.Println(strings.Replace("oink oink oink", "oink", "moo", -1))
```

Output:

```
oinky oinky oink
moo moo moo
```

func Map

```
func Map(mapping func(rune) rune, s string) string
```

将s的每一个unicode码值r都替换为mapping(r)，返回这些新码值组成的字符串拷贝。如果mapping返回一个负值，将会丢弃该码值而不会被替换。（返回值中对应位置将没有码值）

Example

```
rot13 := func(r rune) rune {
    switch {
    case r >= 'A' && r <= 'Z':
        return 'A' + (r-'A'+13)%26
    case r >= 'a' && r <= 'z':
        return 'a' + (r-'a'+13)%26
    }
    return r
}
fmt.Println(strings.Map(rot13, "'Twas brillig and the slithy gophe
```

Output:

```
'Gjnf oevyyvt naq gur fyvgul tbcure...
```

func Trim

```
func Trim(s string, cutset string) string
```

返回将s前后端所有cutset包含的utf-8码值都去掉的字符串。

Example

```
fmt.Printf("[%q]", strings.Trim(" !!! Achtung! Achtung! !!! ", "! '
```

Output:

```
["Achtung! Achtung"]
```

func TrimSpace

```
func TrimSpace(s string) string
```

返回将s前后端所有空白（unicode.IsSpace指定）都去掉的字符串。

Example

```
fmt.Println(strings.TrimSpace(" \t\n a lone gopher \n\t\r\n"))
```

Output:

```
a lone gopher
```

func TrimFunc

```
func TrimFunc(s string, f func(rune) bool) string
```

返回将s前后端所有满足f的unicode码值都去掉的字符串。

func TrimLeft

```
func TrimLeft(s string, cutset string) string
```

返回将s前端所有cutset包含的utf-8码值都去掉的字符串。

func TrimLeftFunc

```
func TrimLeftFunc(s string, f func(rune) bool) string
```

返回将s前端所有满足f的unicode码值都去掉的字符串。

func TrimPrefix

```
func TrimPrefix(s, prefix string) string
```

返回去除s可能的前缀prefix的字符串。

Example

```
var s = "Goodbye,, world!"
s = strings.TrimPrefix(s, "Goodbye,")
s = strings.TrimPrefix(s, "Howdy,")
fmt.Print("Hello" + s)
```

Output:

```
Hello, world!
```

func TrimRight

```
func TrimRight(s string, cutset string) string
```

返回将s后端所有cutset包含的utf-8码值都去掉的字符串。

func TrimRightFunc

```
func TrimRightFunc(s string, f func(rune) bool) string
```

返回将s后端所有满足f的unicode码值都去掉的字符串。

func TrimSuffix

```
func TrimSuffix(s, suffix string) string
```

返回去除s可能的后缀suffix的字符串。

Example

```
var s = "Hello, goodbye, etc!"
s = strings.TrimSuffix(s, "goodbye, etc!")
s = strings.TrimSuffix(s, "planet")
fmt.Print(s, "world!")
```

Output:

```
Hello, world!
```

func Fields

```
func Fields(s string) []string
```

返回将字符串按照空白（unicode.IsSpace确定，可以是一到多个连续的空白字符）分割的多个字符串。如果字符串全部是空白或者是空字符串的话，会返回空切片。

Example

```
fmt.Printf("Fields are: %q", strings.Fields("  foo bar  baz   "))
```

Output:

```
Fields are: ["foo" "bar" "baz"]
```

func FieldsFunc

```
func FieldsFunc(s string, f func(rune) bool) []string
```

类似Fields，但使用函数f来确定分割符（满足f的unicode码值）。如果字符串全部是分隔符或者是空字符串的话，会返回空切片。

Example

```
f := func(c rune) bool {
    return !unicode.IsLetter(c) && !unicode.IsNumber(c)
}
fmt.Printf("Fields are: %q", strings.FieldsFunc("  foo1;bar2,baz3.
```

Output:

```
Fields are: ["foo1" "bar2" "baz3"]
```

func Split

```
func Split(s, sep string) []string
```

用去掉s中出现的sep的方式进行分割，会分割到结尾，并返回生成的所有片段组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个字符串。

Example

```
fmt.Printf("%q\n", strings.Split("a,b,c", ","))
fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a
fmt.Printf("%q\n", strings.Split(" xyz ", ""))
fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
```

Output:

```
["a" "b" "c"]
["" "man " "plan " "canal panama"]
[" " "x" "y" "z" " "]
[""]
```

func SplitN

```
func SplitN(s, sep string, n int) []string
```

用去掉s中出现的sep的方式进行分割，会分割到结尾，并返回生成的所有片段组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个字符串。参数n决定返回的切片的数目：

```
n > 0 : 返回的切片最多n个子字符串；最后一个子字符串包含未进行切割的部分。
n == 0: 返回nil
n < 0 : 返回所有的子字符串组成的切片
```

Example

```
fmt.Printf("%q\n", strings.SplitN("a,b,c", ",", 2))
z := strings.SplitN("a,b,c", ",", 0)
fmt.Printf("%q (nil = %v)\n", z, z == nil)
```

Output:

```
["a" "b,c"]
[] (nil = true)
```

func SplitAfter

```
func SplitAfter(s, sep string) []string
```

用从s中出现的sep后面切断的方式进行分割，会分割到结尾，并返回生成的所有片段组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个字符串。

Example

```
fmt.Printf("%q\n", strings.SplitAfter("a,b,c", ","))
```

Output:

```
["a," "b," "c"]
```

func SplitAfterN

```
func SplitAfterN(s, sep string, n int) []string
```

用从s中出现的sep后面切断的方式进行分割，会分割到结尾，并返回生成的所有片段组成的切片（每一个sep都会进行一次切割，即使两个sep相邻，也会进行两次切割）。如果sep为空字符，Split会将s切分成每一个unicode码值一个字符串。参数n决定返回的切片的数目：

```
n > 0 : 返回的切片最多n个子字符串；最后一个子字符串包含未进行切割的部分。
n == 0: 返回nil
n < 0 : 返回所有的子字符串组成的切
```

Example

```
fmt.Printf("%q\n", strings.SplitAfterN("a,b,c", ",", 2))
```

Output:

```
["a," "b,c"]
```

func Join

```
func Join(a []string, sep string) string
```

将一系列字符串连接为一个字符串，之间用sep来分隔。

Example

```
s := []string{"foo", "bar", "baz"}
fmt.Println(strings.Join(s, ", "))
```

Output:

```
foo, bar, baz
```

type Reader

```
type Reader struct {
    // 内含隐藏或非导出字段
}
```

Reader类型通过从一个字符串读取数据，实现了io.Reader、io.Seeker、io.ReaderAt、io.WriterTo、io.ByteScanner、io.RuneScanner接口。

func NewReader

```
func NewReader(s string) *Reader
```

NewReader创建一个从s读取数据的Reader。本函数类似bytes.NewBufferString，但是更有效率，且为只读的。

func (*Reader) Len

```
func (r *Reader) Len() int
```

Len返回r包含的字符串还没有被读取的部分。

func (*Reader) Read

```
func (r *Reader) Read(b []byte) (n int, err error)
```

func (*Reader) ReadByte

```
func (r *Reader) ReadByte() (b byte, err error)
```

func (*Reader) UnreadByte

```
func (r *Reader) UnreadByte() error
```

func (*Reader) ReadRune

```
func (r *Reader) ReadRune() (ch rune, size int, err error)
```

func (*Reader) UnreadRune

```
func (r *Reader) UnreadRune() error
```

func (*Reader) Seek

```
func (r *Reader) Seek(offset int64, whence int) (int64, error)
```

Seek实现了io.Seeker接口。

func (*Reader) ReadAt

```
func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)
```

func (*Reader) WriteTo

```
func (r *Reader) WriteTo(w io.Writer) (n int64, err error)
```

WriteTo实现了io.WriterTo接口。

type Replacer

```
type Replacer struct {
    // 内含隐藏或非导出字段
}
```

Replacer类型进行一系列字符串的替换。

func NewReplacer

```
func NewReplacer(oldnew ...string) *Replacer
```

使用提供的多组old、new字符串对创建并返回一个*Replacer。替换是依次进行的，匹配时不会重叠。

Example

```
r := strings.NewReplacer("<", "&lt;", ">", "&gt;")
fmt.Println(r.Replace("This is <b>HTML</b>!"))
```

Output:

```
This is &lt;b&gt;HTML&lt;/b&gt;!
```

func (*Replacer) Replace

```
func (r *Replacer) Replace(s string) string
```

Replace返回s的所有替换进行完后的拷贝。

func (*Replacer) WriteString

```
func (r *Replacer) WriteString(w io.Writer, s string) (n int, err e
```

WriteString向w中写入s的所有替换进行完后的拷贝。

package sync

```
import "sync"
```

sync包提供了基本的同步基元，如互斥锁。除了Once和WaitGroup类型，大部分都是适用于低水平程序线程，高水平的同步使用channel通信更好一些。

本包的类型的值不应被拷贝。

Index

Examples

type Locker

```
type Locker interface {
    Lock()
    Unlock()
}
```

Locker接口代表一个可以加锁和解锁的对象。

type Once

```
type Once struct {
    // 包含隐藏或非导出字段
}
```

Once是只执行一次动作的对象。

Example

```
var once sync.Once
onceBody := func() {
    fmt.Println("Only once")
}
done := make(chan bool)
for i := 0; i < 10; i++ {
    go func() {
        once.Do(onceBody)
        done <- true
    }()
}
for i := 0; i < 10; i++ {
    <-done
}
```

Output:

```
Only once
```

func (*Once) Do

```
func (o *Once) Do(f func())
```

Do方法当且仅当第一次被调用时才执行函数f。换句话说，给定变量：

```
var once Once
```

如果once.Do(f)被多次调用，只有第一次调用会执行f，即使f每次调用Do 提供的f值不同。需要给每个要执行仅一次的函数都建立一个Once类型的实例。

Do用于必须刚好运行一次的初始化。因为f是没有参数的，因此可能需要使用闭包来提供给Do方法调用：

```
config.once.Do(func() { config.init(filename) })
```

因为只有f返回后Do方法才会返回，f若引起了Do的调用，会导致死锁。

type Mutex

```
type Mutex struct {
    // 包含隐藏或非导出字段
}
```

Mutex是一个互斥锁，可以创建为其他结构体的字段；零值为解锁状态。Mutex类型的锁和线程无关，可以由不同的线程加锁和解锁。

func (*Mutex) Lock

```
func (m *Mutex) Lock()
```

Lock方法锁住m，如果m已经加锁，则阻塞直到m解锁。

func (*Mutex) Unlock

```
func (m *Mutex) Unlock()
```

Unlock方法解锁m，如果m未加锁会导致运行时错误。锁和线程无关，可以由不同的线程加锁和解锁。

type RWMutex

```
type RWMutex struct {
    // 包含隐藏或非导出字段
}
```

RWMutex是读写互斥锁。该锁可以被同时多个读取者持有或唯一个写入者持有。RWMutex可以创建为其他结构体的字段；零值为解锁状态。RWMutex类型的锁也和线程无关，可以由不同的线程加读取锁/写入和解读取锁/写入锁。

func (*RWMutex) Lock

```
func (rw *RWMutex) Lock()
```

Lock方法将rw锁定为写入状态，禁止其他线程读取或者写入。

func (*RWMutex) Unlock

```
func (rw *RWMutex) Unlock()
```

Unlock方法解除rw的写入锁状态，如果m未加写入锁会导致运行时错误。

func (*RWMutex) RLock

```
func (rw *RWMutex) RLock()
```

RLock方法将rw锁定为读取状态，禁止其他线程写入，但不禁止读取。

func (*RWMutex) RUnlock

```
func (rw *RWMutex) RUnlock()
```

Runlock方法解除rw的读取锁状态，如果m未加读取锁会导致运行时错误。

func (*RWMutex) RLocker

```
func (rw *RWMutex) RLocker() Locker
```

Rlocker方法返回一个互斥锁，通过调用rw.Rlock和rw.Runlock实现了Locker接口。

type Cond

```
type Cond struct {
    // 在观测或更改条件时L会冻结
    L Locker
    // 包含隐藏或非导出字段
}
```

Cond实现了一个条件变量，一个线程集合地，供线程等待或者宣布某事件的发生。

每个Cond实例都有一个相关的锁（一般是*Mutex或*RWMutex类型的值），它必须在改变条件时或者调用Wait方法时保持锁定。Cond可以创建为其他结构体的字段，Cond在开始使用后不能被拷贝。

func NewCond

```
func NewCond(l Locker) *Cond
```

使用锁l创建一个*Cond。

func (*Cond) Broadcast

```
func (c *Cond) Broadcast()
```

Broadcast唤醒所有等待c的线程。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。

func (*Cond) Signal

```
func (c *Cond) Signal()
```

Signal唤醒等待c的一个线程（如果存在）。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。

func (*Cond) Wait

```
func (c *Cond) Wait()
```

Wait自行解锁c.L并阻塞当前线程，在之后线程恢复执行时，Wait方法会在返回前锁定c.L。和其他系统不同，Wait除非被Broadcast或者Signal唤醒，不会主动返回。

因为线程中Wait方法是第一个恢复执行的，而此时c.L未加锁。调用者不应假设Wait恢复时条件已满足，相反，调用者应在循环中等待：

```
c.L.Lock()
for !condition() {
    c.Wait()
}
... make use of condition ...
c.L.Unlock()
```

type WaitGroup

```
type WaitGroup struct {
    // 包含隐藏或非导出字段
}
```

WaitGroup用于等待一组线程的结束。父线程调用Add方法来设定应等待的线程的数量。每个被等待的线程在结束时应调用Done方法。同时，主线程里可以调用Wait方法阻塞至所有线程结束。

Example

```
var wg sync.WaitGroup
var urls = []string{
    "http://www.golang.org/",
    "http://www.google.com/",
    "http://www.somestupidname.com/",
}
for _, url := range urls {
    // Increment the WaitGroup counter.
    wg.Add(1)
    // Launch a goroutine to fetch the URL.
    go func(url string) {
        // Decrement the counter when the goroutine completes.
        defer wg.Done()
        // Fetch the URL.
        http.Get(url)
    }(url)
}
// Wait for all HTTP fetches to complete.
wg.Wait()
```

func (*WaitGroup) Add

```
func (wg *WaitGroup) Add(delta int)
```

Add方法向内部计数加上delta，delta可以是负数；如果内部计数器变为0，Wait方法阻塞等待的所有线程都会释放，如果计数器小于0，方法panic。注意Add加上正数的调用应在Wait之前，否则Wait可能只会等待很少的线程。一般来说本方法应在创建新的线程或者其他应等待的事件之前调用。

func (*WaitGroup) Done

```
func (wg *WaitGroup) Done()
```

Done方法减少WaitGroup计数器的值，应在线程的最后执行。

func (*WaitGroup) Wait

```
func (wg *WaitGroup) Wait()
```

Wait方法阻塞直到WaitGroup计数器减为0。

type Pool

```
type Pool struct {
    // 可选参数New指定一个函数在Get方法可能返回nil时来生成一个值
    // 该参数不能在调用Get方法时被修改
    New func() interface{}
    // 包含隐藏或非导出字段
}
```

Pool是一个可以分别存取的临时对象的集合。

Pool中保存的任何item都可能随时不做通告的释放掉。如果Pool持有该对象的唯一引用，这个item就可能被回收。

Pool可以安全的被多个线程同时使用。

Pool的目的是缓存申请但未使用的item用于之后的重用，以减轻GC的压力。也就是说，让创建高效而线程安全的空闲列表更容易。但Pool并不适用于所有空闲列表。

Pool的合理用法是用于管理一组静静的被多个独立并发线程共享并可能重用的临时item。Pool提供了让多个线程分摊内存申请消耗的方法。

Pool的一个好例子在fmt包里。该Pool维护一个动态大小的临时输出缓存仓库。该仓库会在过载（许多线程活跃的打印时）增大，在沉寂时缩小。

另一方面，管理着短寿命对象的空闲列表不适合使用Pool，因为这种情况下内存申请消耗不能很好的分配。这时应该由这些对象自己实现空闲列表。

func (*Pool) Get

```
func (p *Pool) Get() interface{}
```

Get方法从池中选择任意一个item，删除其在池中的引用计数，并提供给调用者。Get方法也可能选择无视内存池，将其当作空的。调用者不应认为Get的返回这和传递给Put的值之间有任何关系。

假使Get方法没有取得item：如p.New非nil，Get返回调用p.New的结果；否则返回nil。

func (*Pool) Put

```
func (p *Pool) Put(x interface{})
```

Put方法将x放入池中。

package atomic

```
import "sync/atomic"
```

atomic包提供了底层的原子级内存操作，对于同步算法的实现很有用。

这些函数必须谨慎地保证正确使用。除了某些特殊的底层应用，使用通道或者sync包的函数/类型实现同步更好。

应通过通信来共享内存，而不通过共享内存实现通信。

被SwapT系列函数实现的交换操作，在原子性上等价于：

```
old = *addr
*addr = new
return old
```

CompareAndSwapT系列函数实现的比较-交换操作，在原子性上等价于：

```
if *addr == old {
    *addr = new
    return true
}
return false
```

AddT 系列函数实现加法操作，在原子性上等价于：

```
*addr += delta
return *addr
```

LoadT和StoreT系列函数实现的加载和保持操作，在原子性上等价于："return *addr"和"*addr = val"。

Index

func LoadInt32

```
func LoadInt32(addr *int32) (val int32)
```

LoadInt32原子性的获取*addr的值。

func LoadInt64

```
func LoadInt64(addr *int64) (val int64)
```

LoadInt64原子性的获取*addr的值。

func LoadUint32

```
func LoadUint32(addr *uint32) (val uint32)
```

LoadUint32原子性的获取*addr的值。

func LoadUint64

```
func LoadUint64(addr *uint64) (val uint64)
```

LoadUint64原子性的获取*addr的值。

func LoadUintptr

```
func LoadUintptr(addr *uintptr) (val uintptr)
```

LoadUintptr原子性的获取*addr的值。

func LoadPointer

```
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
```

LoadPointer原子性的获取*addr的值。

func StoreInt32

```
func StoreInt32(addr *int32, val int32)
```

StoreInt32原子性的将val的值保存到*addr。

func StoreInt64

```
func StoreInt64(addr *int64, val int64)
```

StoreInt64原子性的将val的值保存到*addr。

func StoreUint32

```
func StoreUint32(addr *uint32, val uint32)
```

StoreUint32原子性的将val的值保存到*addr。

func StoreUint64

```
func StoreUint64(addr *uint64, val uint64)
```

StoreUint64原子性的将val的值保存到*addr。

func StoreUintptr

```
func StoreUintptr(addr *uintptr, val uintptr)
```

StoreUintptr原子性的将val的值保存到*addr。

func StorePointer

```
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
```

StorePointer原子性的将val的值保存到*addr。

func AddInt32

```
func AddInt32(addr *int32, delta int32) (new int32)
```

AddInt32原子性的将val的值添加到*addr并返回新值。

func AddInt64

```
func AddInt64(addr *int64, delta int64) (new int64)
```

AddInt64原子性的将val的值添加到*addr并返回新值。

func AddUint32

```
func AddUint32(addr *uint32, delta uint32) (new uint32)
```

AddUint32原子性的将val的值添加到*addr并返回新值。

如要减去一个值c，调用AddUint32(&x, ^uint32(c-1))；特别的，让x减1，调用AddUint32(&x, ^uint32(0))。

func AddUint64

```
func AddUint64(addr *uint64, delta uint64) (new uint64)
```

AddUint64原子性的将val的值添加到*addr并返回新值。

如要减去一个值c，调用AddUint64(&x, ^uint64(c-1))；特别的，让x减1，调用AddUint64(&x, ^uint64(0))。

func AddUintptr

```
func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)
```

AddUintptr原子性的将val的值添加到*addr并返回新值。

func SwapInt32

```
func SwapInt32(addr *int32, new int32) (old int32)
```

SwapInt32原子性的将新值保存到*addr并返回旧值。

func SwapInt64

```
func SwapInt64(addr *int64, new int64) (old int64)
```

SwapInt64原子性的将新值保存到*addr并返回旧值。

func SwapUint32

```
func SwapUint32(addr *uint32, new uint32) (old uint32)
```

SwapUint32原子性的将新值保存到*addr并返回旧值。

func SwapUint64

```
func SwapUint64(addr *uint64, new uint64) (old uint64)
```

SwapUint64原子性的将新值保存到*addr并返回旧值。

func SwapUintptr

```
func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)
```

SwapUintptr原子性的将新值保存到*addr并返回旧值。

func SwapPointer

```
func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old uns
```

SwapPointer原子性的将新值保存到*addr并返回旧值。

func CompareAndSwapInt32

```
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool
```

CompareAndSwapInt32原子性的比较*addr和old，如果相同则将new赋值给*addr并返回真。

func CompareAndSwapInt64

```
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped boo]
```

CompareAndSwapInt64原子性的比较*addr和old，如果相同则将new赋值给*addr并返回真。

func CompareAndSwapUint32

```
func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped b
```

CompareAndSwapUint32原子性的比较*addr和old，如果相同则将new赋值给*addr并返回真。

func CompareAndSwapUint64

```
func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped b
```

CompareAndSwapUint64原子性的比较*addr和old，如果相同则将new赋值给*addr并返回真。

func CompareAndSwapUintptr

```
func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swappe
```

CompareAndSwapUintptr原子性的比较*addr和old，如果相同则将new赋值给*addr并返回真。

func CompareAndSwapPointer

```
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Po
```

CompareAndSwapPointer原子性的比较*addr和old，如果相同则将new赋值给*addr并返回真。

Bugs

☞ 在x86-32构架芯片上，64位函数使用的指令在Pentium MMX之前是不可用的；
在非Linux的ARM芯片上，64位函数使用的指令在ARMv6k core之前是不可用的。
不管是ARM又或x86-32芯片，安排原子性访问的64位word的64位对齐都是调用者的责任。可以依靠全局变量或申请的切片/结构体的第一个word来实现64位对齐。

package text

package scanner

`import "text/scanner"`

scanner包提供对utf-8文本的token扫描服务。它会从一个io.Reader获取utf-8文本，通过对Scan方法的重复调用获取一个个token。为了兼容已有的工具，NUL字符不被接受。如果第一个字符是表示utf-8编码格式的BOM标记，会自动忽略该标记。

一般Scanner会跳过空白和Go注释，并会识别所有go语言规格的字面量。它可以定制为只识别这些字面量的一个子集，也可以识别不同的空白字符。

基本使用模式：

```
var s scanner.Scanner
s.Init(src)
tok := s.Scan()
for tok != scanner.EOF {
    // do something with tok
    tok = s.Scan()
}
```

Index

Constants

```
const (
    ScanIdents     = 1 << -Ident
    ScanInts       = 1 << -Int
    ScanFloats     = 1 << -Float // 包括整数
    ScanChars      = 1 << -Char
    ScanStrings    = 1 << -String
    ScanRawStrings = 1 << -RawString
    ScanComments   = 1 << -Comment
    SkipComments   = 1 << -skipComment // 如设置了ScanComments就视注释
    GoTokens       = ScanIdents | ScanFloats | ScanChars
                   | ScanStrings | ScanRawStrings | ScanComments |
)
```

预定义的状态位，用于控制token的识别。例如，如要设置Scanner只识别标识符、整数、跳过注释，可以将Scanner的状态字段设为：

```
ScanIdents | ScanInts | SkipComments
```

```
const (
    EOF = -(iota + 1)
    Ident
    Int
    Float
    Char
    String
    RawString
    Comment
)
```

扫描的结果是上面的一个token或者一个Unicode字符。

```
const GoWhitespace = 1<<'\t' | 1<<'\n' | 1<<'\r' | 1<<' '
```

GoWhitespace是一个Scanner的Whitespace字段的默认值，该值确定go的空白字符。

func TokenString

```
func TokenString(tok rune) string
```

TokenString返回一个token或unicode码值的可打印的字符串表示。

type Position

```
type Position struct {
    Filename string // 文件名（如果存在）
    Offset   int    // 偏移量，从0开始
    Line     int    // 行号，从1开始
    Column   int    // 列号，从1开始（每行第几个字符）
}
```

代表资源里的一个位置。

func (*Position) IsValid

```
func (pos *Position) IsValid() bool
```

IsValid返回所处的位置是否合法。

func (Position) String

```
func (pos Position) String() string
```

type Scanner

```
type Scanner struct {
    // 每一次出现错误时都会调用该函数；如果Error为nil，则会将错误报告到os.St
    Error func(s *Scanner, msg string)
    // 每一次出现错误时，ErrorCount++
    ErrorCount int
    // 控制那些token被识别。如要识别整数，就将Mode的ScanInts位设为1。随时都
    Mode uint
    // 控制那些字符识别为空白。如果要将一个码值小于32的字符视为空白，只需将码值
    // 空格码值是32，大于32的位设为1的行为未定义。随时都可以修改Whitespace。
    Whitespace uint64
    // 最近一次扫描到的token的开始位置，由Scan方法设定
    // 调用Init或Next方法会使位置无效（Line==0），Scanner不会操作Position
    // 如果发生错误且Position不合法，此时扫描位置不在token内，应调用Pos获取
    Position
    // 内含隐藏或非导出字段
}
```

Scanner类型实现了token和unicode字符（从io.Reader中）的读取。

func (*Scanner) Init

```
func (s *Scanner) Init(src io.Reader) *Scanner
```

Init使用src创建一个Scanner，并将Error设为nil，ErrorCount设为0，Mode设为GoTokens，Whitespace 设为GoWhitespace。

func (*Scanner) Pos

```
func (s *Scanner) Pos() (pos Position)
```

Pos方法返回上一次调用Next或Scan方法后读取结束时的位置。

func (*Scanner) Peek

```
func (s *Scanner) Peek() rune
```

Peek方法返回资源的下一个unicode字符而不移动扫描位置。如果扫描位置在资源的结尾会返回EOF。

func (*Scanner) Next

```
func (s *Scanner) Next() rune
```

Next读取并返回下一个unicode字符。到达资源结尾时会返回EOF。如果s.Error非nil，本方法会调用该字段汇报错误；否则将错误信息发送到os.Stderr。Next不会更新Scanner的Position字段，请使用Pos方法获取当前位置。

func (*Scanner) Scan

```
func (s *Scanner) Scan() rune
```

Scan方法从资源读取下一个token或者unicode字符并返回它。本方法只会识别Mode字段指定的token种类。如果到达资源结尾会返回EOF。如果s.Error非nil，本方法会调用该字段汇报错误；否则将错误信息发送到os.Stderr。

func (*Scanner) TokenText

```
func (s *Scanner) TokenText() string
```

TokenText方法返回最近一次扫描的token对应的字符串。应该在Scan方法后调用。

package tabwriter

```
import "text/tabwriter"
```

tabwriter包实现了写入过滤器（tabwriter.Writer），可以将输入的缩进修正为正确的对齐文本。

本包采用的Elastic Tabstops算法参见 http://nickgravgaard.com/elastictabstops/index.html

Index

Examples

Constants

```
const (
    // 忽略html标签，并将字符实体（以'&'开始，以';'结束）视为单字符
    FilterHTML uint = 1 << iota
    // 将转义后文本两端的转义字符去掉
    StripEscape
    // 强制单元格右对齐，默认是左对齐的
    AlignRight
    // 剔除空行
    DiscardEmptyColumns
    // 使用tab而不是空格进行缩进
    TabIndent
    // 在格式化后在每一列两侧添加'|'并忽略空行
    Debug
)
```

这些标志用于控制格式化。

```
const Escape = '\xff'
```

用于包围转义字符，避免该字符被转义；例如字符串"Ignore this tab: \xff\t\xff"中的'\t'不被转义，不结束单元；格式化时Escape视为长度1的单字符。

选择'\xff'是因为该字符不能出现在合法的utf-8序列里。

type Writer

```
type Writer struct {
    // 内含隐藏或非导出字段
}
```

Writer是一个过滤器，会在输入的tab划分的列进行填充，在输出中对齐它们。

它会将输入的序列视为utf-8编码的文本，包含一系列被垂直制表符、水平制表符、换行符、回车符分割的单元。临近的单元组成一列，根据需要填充空格使所有的单元有相同的宽度，高效对齐各列。它假设所有的字符都有相同的宽度，除了tab，tab宽度应该被指定。注意单元以tab截止，而不是被tab分隔，行最后的非tab文本不被视为列的单元。

Writer假设所有的unicode字符有着同样的宽度，这一点其实在很多字体里是错误的。

如果设置了DiscardEmptyColumns，以垂直制表符结尾的空列会被丢弃，水平制表符截止的空列则不会被影响。

如果设置了FilterHTML，HTML标签和实体会被放过，标签宽度视为0，实体宽度视为1。文本段可能被转义字符包围，此时tabwriter不会修改该文本段，不会打断段中的任何tab或换行。

如果设置了StripEscape，则不会计算转义字符的宽度（输出中也会去除转义字符）。

进纸符'\f'被视为换行，但也会截止当前行的所有列（有效的刷新）；除非在HTML标签内或者在转义文本段内，输出中'\f'都被作为换行。

Writer会在内部缓存输入以便有效的对齐，调用者必须在写完后执行Flush方法。

func NewWriter

```
func NewWriter(output io.Writer, minwidth, tabwidth, padding int, p
```

创建并初始化一个tabwriter.Writer，参数用法和Init函数类似。

func (*Writer) Init

```
func (b *Writer) Init(output io.Writer, minwidth, tabwidth, paddinç
```

初始化一个Writer，第一个参数指定格式化后的输出目标，其余的参数控制格式化：

```
minwidth 最小单元长度
tabwidth tab字符的宽度
padding  计算单元宽度时会额外加上它
padchar  用于填充的ASCII字符，
         如果是'\t'，则Writer会假设tabwidth作为输出中tab的宽度，且单元必然
flags    格式化控制
```

Example

```
w := new(tabwriter.Writer)
// Format in tab-separated columns with a tab stop of 8.
w.Init(os.Stdout, 0, 8, 0, '\t', 0)
fmt.Fprintln(w, "a\tb\tc\td\t.")
fmt.Fprintln(w, "123\t12345\t1234567\t123456789\t.")
fmt.Fprintln(w)
w.Flush()
// Format right-aligned in space-separated columns of minimal widtl
// and at least one blank of padding (so wider column entries do nc
// touch each other).
w.Init(os.Stdout, 5, 0, 1, ' ', tabwriter.AlignRight)
fmt.Fprintln(w, "a\tb\tc\td\t.")
fmt.Fprintln(w, "123\t12345\t1234567\t123456789\t.")
fmt.Fprintln(w)
w.Flush()
```

Output:

```
a    b    c    d        .
123  12345  1234567  123456789  .
    a     b       c         d.
  123 12345 1234567 123456789.
```

func (*Writer) Write

```
func (b *Writer) Write(buf []byte) (n int, err error)
```

将buf写入b，实现io.Writer接口，只有在写入底层输出流是才可能发生并返回错误。

func (*Writer) Flush

```
func (b *Writer) Flush() (err error)
```

在最后一次调用Write后，必须调用Flush方法以清空缓存，并将格式化对齐后的文本写入生成时提供的output中。

package template

```
import "text/template"
```

template包实现了数据驱动的用于生成文本输出的模板。

如果要生成HTML格式的输出，参见html/template包，该包提供了和本包相同的接口，但会自动将输出转化为安全的HTML格式输出，可以抵抗一些网络攻击。

通过将模板应用于一个数据结构（即该数据结构作为模板的参数）来执行，来获得输出。模板中的注释引用数据接口的元素（一般如结构体的字段或者字典的键）来控制执行过程和获取需要呈现的值。模板执行时会遍历结构并将指针表示为'.'（称之为"dot"）指向运行过程中数据结构的当前位置的值。

用作模板的输入文本必须是utf-8编码的文本。"Action"—数据运算和控制单位—由"和"界定；在Action之外的所有文本都不做修改的拷贝到输出中。Action内部不能有换行，但注释可以有换行。

经解析生成模板后，一个模板可以安全的并发执行。

下面是一个简单的例子，可以打印"17 of wool"。

```
type Inventory struct {
    Material string
    Count    uint
}
sweaters := Inventory{"wool", 17}
tmpl, err := template.New("test").Parse("{{.Count}} of {{.Material]
if err != nil { panic(err) }
err = tmpl.Execute(os.Stdout, sweaters)
if err != nil { panic(err) }
```

更复杂的例子在下面。

Actions

下面是一个action（动作）的列表。"Arguments"和"pipelines"代表数据的执行结果，细节定义在后面。

```
{{/* a comment */}}
    注释，执行时会忽略。可以多行。注释不能嵌套，并且必须紧贴分界符始止，就像这모
{{pipeline}}
    pipeline的值的默认文本表示会被拷贝到输出里。
{{if pipeline}} T1 {{end}}
    如果pipeline的值为empty，不产生输出，否则输出T1执行结果。不改变dot的值。
    Empty值包括false、0、任意nil指针或者nil接口，任意长度为0的数组、切片、字
{{if pipeline}} T1 {{else}} T0 {{end}}
    如果pipeline的值为empty，输出T0执行结果，否则输出T1执行结果。不改变dot的
{{if pipeline}} T1 {{else if pipeline}} T0 {{end}}
    用于简化if-else链条，else action可以直接包含另一个if；等价于：
        {{if pipeline}} T1 {{else}}{{if pipeline}} T0 {{end}}{{end}
{{range pipeline}} T1 {{end}}
    pipeline的值必须是数组、切片、字典或者通道。
    如果pipeline的值其长度为0，不会有任何输出；
    否则dot依次设为数组、切片、字典或者通道的每一个成员元素并执行T1；
    如果pipeline的值为字典，且键可排序的基本类型，元素也会按键的顺序排序。
{{range pipeline}} T1 {{else}} T0 {{end}}
    pipeline的值必须是数组、切片、字典或者通道。
    如果pipeline的值其长度为0，不改变dot的值并执行T0；否则会修改dot并执行T1
{{template "name"}}
    执行名为name的模板，提供给模板的参数为nil，如模板不存在输出为""
{{template "name" pipeline}}
    执行名为name的模板，提供给模板的参数为pipeline的值。
{{with pipeline}} T1 {{end}}
    如果pipeline为empty不产生输出，否则将dot设为pipeline的值并执行T1。不修
{{with pipeline}} T1 {{else}} T0 {{end}}
    如果pipeline为empty，不改变dot并执行T0，否则dot设为pipeline的值并执行
```

Arguments

参数代表一个简单的，由下面的某一条表示的值：

```
- go语法的布尔值、字符串、字符、整数、浮点数、虚数、复数，视为无类型字面常数，字
- 关键字nil，代表一个go的无类型的nil值
- 字符'.'（句点，用时不加单引号），代表dot的值
- 变量名，以美元符号起始加上（可为空的）字母和数字构成的字符串，如：$piOver2和
  执行结果为变量的值，变量参见下面的介绍
- 结构体数据的字段名，以句点起始，如：.Field；
  执行结果为字段的值，支持链式调用：.Field1.Field2；
  字段也可以在变量上使用（包括链式调用）：$x.Field1.Field2；
- 字典类型数据的键名；以句点起始，如：.Key；
  执行结果是该键在字典中对应的成员元素的值；
  键也可以和字段配合做链式调用，深度不限：.Field1.Key1.Field2.Key2；
  虽然键也必须是字母和数字构成的标识字符串，但不需要以大写字母起始；
  键也可以用于变量（包括链式调用）：$x.key1.key2；
- 数据的无参数方法名，以句点为起始，如：.Method；
  执行结果为dot调用该方法的返回值，dot.Method()；
  该方法必须有1到2个返回值，如果有2个则后一个必须是error接口类型；
  如果有2个返回值的方法返回的error非nil，模板执行会中断并返回给调用模板执行者
  方法可和字段、键配合做链式调用，深度不限：.Field1.Key1.Method1.Field2.Ke
  方法也可以在变量上使用（包括链式调用）：$x.Method1.Field；
- 无参数的函数名，如：fun；
  执行结果是调用该函数的返回值fun()；对返回值的要求和方法一样；函数和函数名细节
- 上面某一条的实例加上括弧（用于分组）
  执行结果可以访问其字段或者键对应的值：
      print (.F1 arg1) (.F2 arg2)
      (.StructValuedMethod "arg").Field
```

Arguments可以是任何类型：如果是指针，在必要时会自动表示为指针指向的值；如果执行结果生成了一个函数类型的值，如结构体的函数类型字段，该函数不会自动调用，但可以在if等action里视为真。如果要调用它，使用call函数，参见下面。

Pipeline是一个（可能是链状的）command序列。Command可以是一个简单值（argument）或者对函数或者方法的（可以有多个参数的）调用：

```
Argument
    执行结果是argument的执行结果
.Method [Argument...]
    方法可以独立调用或者位于链式调用的末端，不同于链式调用中间的方法，可以使用参
    执行结果是使用给出的参数调用该方法的返回值：dot.Method(Argument1, etc.
functionName [Argument...]
    执行结果是使用给定的参数调用函数名指定的函数的返回值：function(Argument1
```

Pipelines

pipeline通常是将一个command序列分割开，再使用管道符'|'连接起来（但不使用管道符的command序列也可以视为一个管道）。在一个链式的pipeline里，每个command的结果都作为下一个command的最后一个参数。pipeline最后一个command的输出作为整个管道执行的结果。

command的输出可以是1到2个值，如果是2个后一个必须是error接口类型。如果error类型返回值非nil，模板执行会中止并将该错误返回给执行模板的调用者。

Variables

Action里可以初始化一个变量来捕获管道的执行结果。初始化语法如下：

```
$variable := pipeline
```

其中$variable是变量的名字。声明变量的action不会产生任何输出。

如果"range" action初始化了1个变量，该变量设置为迭代器的每一个成员元素，如果初始化了逗号分隔的2个变量：

```
range $index, $element := pipeline
```

这时，$index和$element分别设置为数组/切片的索引或者字典的键，以及对应的成员元素。注意这和go range从句只有一个参数时设置为索引/键不同！

一个变量的作用域只到声明它的控制结构（"if"、"with"、"range"）的"end"为止，如果不是在控制结构里声明会直到模板结束为止。子模板的调用不会从调用它的位置（作用域）继承变量。

模板开始执行时，$会设置为传递给Execute方法的参数，就是说，dot的初始值。

Examples

下面是一些单行模板，展示了pipeline和变量。所有都生成加引号的单词"output"：

```
{{"\"output\""}}
    字符串常量
{{`"output"`}}
    原始字符串常量
{{printf "%q" "output"}}
    函数调用
{{"output" | printf "%q"}}
    函数调用，最后一个参数来自前一个command的返回值
{{printf "%q" (print "out" "put")}}
    加括号的参数
{{"put" | printf "%s%s" "out" | printf "%q"}}
    玩出花的管道的链式调用
{{"output" | printf "%s" | printf "%q"}}
    管道的链式调用
{{with "output"}}{{printf "%q" .}}{{end}}
    使用dot的with action
{{with $x := "output" | printf "%q"}}{{$x}}{{end}}
    创建并使用变量的with action
{{with $x := "output"}}{{printf "%q" $x}}{{end}}
    将变量使用在另一个action的with action
{{with $x := "output"}}{{$x | printf "%q"}}{{end}}
    以管道形式将变量使用在另一个action的with action
```

Functions

执行模板时，函数从两个函数字典中查找：首先是模板函数字典，然后是全局函数字典。一般不在模板内定义函数，而是使用Funcs方法添加函数到模板里。

预定义的全局函数如下：

```
and
    函数返回它的第一个empty参数或者最后一个参数；
    就是说"and x y"等价于"if x then y else x"；所有参数都会执行；
or
    返回第一个非empty参数或者最后一个参数；
    亦即"or x y"等价于"if x then x else y"；所有参数都会执行；
not
    返回它的单个参数的布尔值的否定
len
    返回它的参数的整数类型长度
index
    执行结果为第一个参数以剩下的参数为索引/键指向的值；
    如"index x 1 2 3"返回x[1][2][3]的值；每个被索引的主体必须是数组、切片
print
    即fmt.Sprint
printf
    即fmt.Sprintf
println
    即fmt.Sprintln
html
    返回其参数文本表示的HTML逸码等价表示。
urlquery
    返回其参数文本表示的可嵌入URL查询的逸码等价表示。
js
    返回其参数文本表示的JavaScript逸码等价表示。
call
    执行结果是调用第一个参数的返回值，该参数必须是函数类型，其余参数作为调用该
    如"call .X.Y 1 2"等价于go语言里的dot.X.Y(1, 2)；
    其中Y是函数类型的字段或者字典的值，或者其他类似情况；
    call的第一个参数的执行结果必须是函数类型的值（和预定义函数如print明显不同
    该函数类型值必须有1到2个返回值，如果有2个则后一个必须是error接口类型；
    如果有2个返回值的方法返回的error非nil，模板执行会中断并返回给调用模板执行
```

布尔函数会将任何类型的零值视为假，其余视为真。

下面是定义为函数的二元比较运算的集合：

```
eq      如果arg1 == arg2则返回真
ne      如果arg1 != arg2则返回真
lt      如果arg1 < arg2则返回真
le      如果arg1 <= arg2则返回真="" gt="" 如果arg1=""> arg2则返回真
ge      如果arg1 >= arg2则返回真
```

为了简化多参数相等检测，eq（只有eq）可以接受2个或更多个参数，它会将第一个参数和其余参数依次比较，返回下式的结果：

```
arg1==arg2 || arg1==arg3 || arg1==arg4 ...
```

（和go的||不一样，不做惰性运算，所有参数都会执行）

比较函数只适用于基本类型（或重定义的基本类型，如"type Celsius float32"）。它们实现了go语言规则的值的比较，但具体的类型和大小会忽略掉，因此任意类型有符号整数值都可以互相比较；任意类型无符号整数值都可以互相比较；等等。但是，整数和浮点数不能互相比较。

Associated templates

每一个模板在创建时都要用一个字符串命名。同时，每一个模板都会和0或多个模板关联，并可以使用它们的名字调用这些模板；这种关联可以传递，并形成一个模板的名字空间。

一个模板可以通过模板调用实例化另一个模板；参见上面的"template" action。name必须是包含模板调用的模板相关联的模板的名字。

Nested template definitions

当解析模板时，可以定义另一个模板，该模板会和当前解析的模板相关联。模板必须定义在当前模板的最顶层，就像go程序里的全局变量一样。

这种定义模板的语法是将每一个子模板的声明放在"define"和"end" action内部。

define action使用给出的字符串常数给模板命名，举例如下：

```
`{{define "T1"}}ONE{{end}}
{{define "T2"}}TWO{{end}}
{{define "T3"}}{{template "T1"}} {{template "T2"}}{{end}}
{{template "T3"}}`
```

它定义了两个模板T1和T2，第三个模板T3在执行时调用这两个模板；最后该模板调用了T3。输出结果是：

```
ONE TWO
```

采用这种方法，一个模板只能从属于一个关联。如果需要让一个模板可以被多个关联查找到；模板定义必须多次解析以创建不同的*Template 值，或者必须使用Clone或AddParseTree方法进行拷贝。

可能需要多次调用Parse函数以集合多个相关的模板；参见ParseFiles和ParseGlob函数和方法，它们提供了简便的途径去解析保存在文件中的存在关联的模板。

一个模板可以直接调用或者通过ExecuteTemplate方法调用指定名字的相关联的模板；我们可以这样调用模板：

```
err := tmpl.Execute(os.Stdout, "no data needed")
if err != nil {
    log.Fatalf("execution failed: %s", err)
}
```

或显式的指定模板的名字来调用：

```
err := tmpl.ExecuteTemplate(os.Stdout, "T2", "no data needed")
if err != nil {
    log.Fatalf("execution failed: %s", err)
}
```

Index

Examples

- Template
- Template (Func)
- Template (Glob)
- Template (Helpers)
- Template (Share)

func HTMLEscape

```
func HTMLEscape(w io.Writer, b []byte)
```

函数向w中写入b的HTML转义等价表示。

func HTMLEscapeString

```
func HTMLEscapeString(s string) string
```

返回s的HTML转义等价表示字符串。

func HTMLEscaper

```
func HTMLEscaper(args ...interface{}) string
```

函数返回其所有参数文本表示的HTML转义等价表示字符串。

func JSEscape

```
func JSEscape(w io.Writer, b []byte)
```

函数向w中写入b的JavaScript转义等价表示。

func JSEscapeString

```
func JSEscapeString(s string) string
```

返回s的JavaScript转义等价表示字符串。

func JSEscaper

```
func JSEscaper(args ...interface{}) string
```

函数返回其所有参数文本表示的JavaScript转义等价表示字符串。

func URLQueryEscaper

```
func URLQueryEscaper(args ...interface{}) string
```

函数返回其所有参数文本表示的可以嵌入URL查询的转义等价表示字符串。

type FuncMap

```
type FuncMap map[string]interface{}
```

FuncMap类型定义了函数名字符串到函数的映射，每个函数都必须有1到2个返回值，如果有2个则后一个必须是error接口类型；如果有2个返回值的方法返回的error非nil，模板执行会中断并返回给调用者该错误。

type Template

```
type Template struct {
    *parse.Tree
    // 内含隐藏或非导出字段
}
```

代表一个解析好的模板，*parse.Tree字段仅仅是暴露给html/template包使用的，因此其他人应该视字段未导出。

Example

```
// Define a template.
const letter = `
Dear {{.Name}},
{{if .Attended}}
It was a pleasure to see you at the wedding.{{else}}
It is a shame you couldn't make it to the wedding.{{end}}
{{with .Gift}}Thank you for the lovely {{.}}.
{{end}}
Best wishes,
Josie
`
// Prepare some data to insert into the template.
type Recipient struct {
    Name, Gift string
    Attended   bool
}
var recipients = []Recipient{
    {"Aunt Mildred", "bone china tea set", true},
    {"Uncle John", "moleskin pants", false},
    {"Cousin Rodney", "", false},
}
// Create a new template and parse the letter into it.
t := template.Must(template.New("letter").Parse(letter))
// Execute the template for each recipient.
for _, r := range recipients {
    err := t.Execute(os.Stdout, r)
    if err != nil {
        log.Println("executing template:", err)
    }
}
```

Output:

```
Dear Aunt Mildred,
It was a pleasure to see you at the wedding.
Thank you for the lovely bone china tea set.
Best wishes,
Josie
Dear Uncle John,
It is a shame you couldn't make it to the wedding.
Thank you for the lovely moleskin pants.
Best wishes,
Josie
Dear Cousin Rodney,
It is a shame you couldn't make it to the wedding.
Best wishes,
Josie
```

Example (Func)

```
// First we create a FuncMap with which to register the function.
funcMap := template.FuncMap{
    // The name "title" is what the function will be called in the
    "title": strings.Title,
}
// A simple template definition to test our function.
// We print the input text several ways:
// - the original
// - title-cased
// - title-cased and then printed with %q
// - printed with %q and then title-cased.
const templateText = `
Input: {{printf "%q" .}}
Output 0: {{title .}}
Output 1: {{title . | printf "%q"}}
Output 2: {{printf "%q" . | title}}
`
// Create a template, add the function map, and parse the text.
tmpl, err := template.New("titleTest").Funcs(funcMap).Parse(templat
if err != nil {
    log.Fatalf("parsing: %s", err)
}
// Run the template to verify the output.
err = tmpl.Execute(os.Stdout, "the go programming language")
if err != nil {
    log.Fatalf("execution: %s", err)
}
```

Output:

```
Input: "the go programming language"
Output 0: The Go Programming Language
Output 1: "The Go Programming Language"
Output 2: "The Go Programming Language"
```

Example (Glob)

```
// Here we create a temporary directory and populate it with our sa
// template definition files; usually the template files would alre
// exist in some location known to the program.
dir := createTestDir([]templateFile{
    // T0.tmpl is a plain template file that just invokes T1.
    {"T0.tmpl", `T0 invokes T1: ({{template "T1"}})`},
    // T1.tmpl defines a template, T1 that invokes T2.
    {"T1.tmpl", `{{define "T1"}}T1 invokes T2: ({{template "T2"}})
    // T2.tmpl defines a template T2.
    {"T2.tmpl", `{{define "T2"}}This is T2{{end}}`},
})
// Clean up after the test; another quirk of running as an example.
defer os.RemoveAll(dir)
// pattern is the glob pattern used to find all the template files.
pattern := filepath.Join(dir, "*.tmpl")
// Here starts the example proper.
// T0.tmpl is the first name matched, so it becomes the starting te
// the value returned by ParseGlob.
tmpl := template.Must(template.ParseGlob(pattern))
err := tmpl.Execute(os.Stdout, nil)
if err != nil {
    log.Fatalf("template execution: %s", err)
}
```

Output:

```
T0 invokes T1: (T1 invokes T2: (This is T2))
```

Example (Helpers)

```
// Here we create a temporary directory and populate it with our sa
// template definition files; usually the template files would alre
// exist in some location known to the program.
dir := createTestDir([]templateFile{
    // T1.tmpl defines a template, T1 that invokes T2.
    {"T1.tmpl", `{{define "T1"}}T1 invokes T2: ({{template "T2"}})
    // T2.tmpl defines a template T2.
    {"T2.tmpl", `{{define "T2"}}This is T2{{end}}`},
})
// Clean up after the test; another quirk of running as an example.
defer os.RemoveAll(dir)
// pattern is the glob pattern used to find all the template files.
pattern := filepath.Join(dir, "*.tmpl")
// Here starts the example proper.
// Load the helpers.
templates := template.Must(template.ParseGlob(pattern))
// Add one driver template to the bunch; we do this with an explici
_, err := templates.Parse("{{define `driver1`}}Driver 1 calls T1: (
if err != nil {
    log.Fatal("parsing driver1: ", err)
}
// Add another driver template.
_, err = templates.Parse("{{define `driver2`}}Driver 2 calls T2: (
if err != nil {
    log.Fatal("parsing driver2: ", err)
}
// We load all the templates before execution. This package does no
// that behavior but html/template's escaping does, so it's a good
err = templates.ExecuteTemplate(os.Stdout, "driver1", nil)
if err != nil {
    log.Fatalf("driver1 execution: %s", err)
}
err = templates.ExecuteTemplate(os.Stdout, "driver2", nil)
if err != nil {
    log.Fatalf("driver2 execution: %s", err)
}
```

Output:

```
Driver 1 calls T1: (T1 invokes T2: (This is T2))
Driver 2 calls T2: (This is T2)
```

Example (Share)

```
// Here we create a temporary directory and populate it with our sa
// template definition files; usually the template files would alre
// exist in some location known to the program.
dir := createTestDir([]templateFile{
    // T0.tmpl is a plain template file that just invokes T1.
    {"T0.tmpl", "T0 ({{.}} version) invokes T1: ({{template `T1`}}
    // T1.tmpl defines a template, T1 that invokes T2\. Note T2 is
    {"T1.tmpl", `{{define "T1"}}T1 invokes T2: ({{template "T2"}})
})
// Clean up after the test; another quirk of running as an example
defer os.RemoveAll(dir)
// pattern is the glob pattern used to find all the template files
pattern := filepath.Join(dir, "*.tmpl")
// Here starts the example proper.
// Load the drivers.
drivers := template.Must(template.ParseGlob(pattern))
// We must define an implementation of the T2 template. First we cl
// the drivers, then add a definition of T2 to the template name sp
// 1\. Clone the helper set to create a new name space from which t
first, err := drivers.Clone()
if err != nil {
    log.Fatal("cloning helpers: ", err)
}
// 2\. Define T2, version A, and parse it.
_, err = first.Parse("{{define `T2`}}T2, version A{{end}}")
if err != nil {
    log.Fatal("parsing T2: ", err)
}
// Now repeat the whole thing, using a different version of T2.
// 1\. Clone the drivers.
second, err := drivers.Clone()
if err != nil {
    log.Fatal("cloning drivers: ", err)
}
// 2\. Define T2, version B, and parse it.
_, err = second.Parse("{{define `T2`}}T2, version B{{end}}")
if err != nil {
    log.Fatal("parsing T2: ", err)
}
// Execute the templates in the reverse order to verify the
// first is unaffected by the second.
err = second.ExecuteTemplate(os.Stdout, "T0.tmpl", "second")
if err != nil {
    log.Fatalf("second execution: %s", err)
}
err = first.ExecuteTemplate(os.Stdout, "T0.tmpl", "first")
if err != nil {
    log.Fatalf("first: execution: %s", err)
}
```

Output:

```
T0 (second version) invokes T1: (T1 invokes T2: (T2, version B))
T0 (first version) invokes T1: (T1 invokes T2: (T2, version A))
```

func Must

```
func Must(t *Template, err error) *Template
```

Must函数用于包装返回(*Template, error)的函数/方法调用，它会在err非nil时panic，一般用于变量初始化：

```
var t = template.Must(template.New("name").Parse("text"))
```

func New

```
func New(name string) *Template
```

创建一个名为name的模板。

func ParseFiles

```
func ParseFiles(filenames ...string) (*Template, error)
```

ParseFiles函数创建一个模板并解析filenames指定的文件里的模板定义。返回的模板的名字是第一个文件的文件名（不含扩展名），内容为解析后的第一个文件的内容。至少要提供一个文件。如果发生错误，会停止解析并返回nil。

func ParseGlob

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob创建一个模板并解析匹配pattern的文件（参见glob规则）里的模板定义。返回的模板的名字是第一个匹配的文件的文件名（不含扩展名），内容为解析后的第一个文件的内容。至少要存在一个匹配的文件。如果发生错误，会停止解析并返回nil。ParseGlob等价于使用匹配pattern的文件的列表为参数调用ParseFiles。

func (*Template) Name

```
func (t *Template) Name() string
```

返回模板t的名字。

func (*Template) Delims

```
func (t *Template) Delims(left, right string) *Template
```

Delims方法用于设置action的分界字符串，应用于之后的Parse、ParseFiles、ParseGlob方法。嵌套模板定义会继承这种分界符设置。空字符串分界符表示相应的默认分界符：。返回值就是t，以便进行链式调用。

func (*Template) Funcs

```
func (t *Template) Funcs(funcMap FuncMap) *Template
```

Funcs方法向模板t的函数字典里加入参数funcMap内的键值对。如果funcMap某个键值对的值不是函数类型或者返回值不符合要求会panic。但是，可以对t函数列表的成员进行重写。方法返回t以便进行链式调用。

func (*Template) Clone

```
func (t *Template) Clone() (*Template, error)
```

Clone方法返回模板的一个副本，包括所有相关联的模板。模板的底层表示树并未拷贝，而是拷贝了命名空间，因此拷贝调用Parse方法不会修改原模板的命名空间。Clone方法用于准备模板的公用部分，向拷贝中加入其他关联模板后再进行使用。

func (*Template) Lookup

```
func (t *Template) Lookup(name string) *Template
```

Lookup方法返回与t关联的名为name的模板，如果没有这个模板返回nil。

func (*Template) Templates

```
func (t *Template) Templates() []*Template
```

Templates方法返回与t相关联的模板的切片，包括t自己。

func (*Template) New

```
func (t *Template) New(name string) *Template
```

New方法创建一个和t关联的名字为name的模板并返回它。这种可以传递的关联允许一个模板使用template action调用另一个模板。

func (*Template) AddParseTree

```
func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Te
```

AddParseTree方法使用name和tree创建一个模板并使它和t相关联。

func (*Template) Parse

```
func (t *Template) Parse(text string) (*Template, error)
```

Parse方法将字符串text解析为模板。嵌套定义的模板会关联到最顶层的t。Parse可以多次调用，但只有第一次调用可以包含空格、注释和模板定义之外的文本。如果后面的调用在解析后仍剩余文本会引发错误、返回nil且丢弃剩余文本；如果解析得到的模板已有相关联的同名模板，会覆盖掉原模板。

func (*Template) ParseFiles

```
func (t *Template) ParseFiles(filenames ...string) (*Template, erro
```

ParseGlob方法解析filenames指定的文件里的模板定义并将解析结果与t关联。如果发生错误，会停止解析并返回nil，否则返回(t, nil)。至少要提供一个文件。

func (*Template) ParseGlob

```
func (t *Template) ParseGlob(pattern string) (*Template, error)
```

ParseFiles方法解析匹配pattern的文件里的模板定义并将解析结果与t关联。如果发生错误，会停止解析并返回nil，否则返回(t, nil)。至少要存在一个匹配的文件。

func (*Template) Execute

```
func (t *Template) Execute(wr io.Writer, data interface{}) (err er
```

Execute方法将解析好的模板应用到data上，并将输出写入wr。如果执行时出现错误，会停止执行，但有可能已经写入wr部分数据。模板可以安全的并发执行。

func (*Template) ExecuteTemplate

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data
```

ExecuteTemplate方法类似Execute，但是使用名为name的t关联的模板产生输出。

package time

```
import "time"
```

time包提供了时间的显示和测量用的函数。日历的计算采用的是公历。

Index

Examples

- Sleep
- Tick
- Time.Format
- Time.Round
- Time.Truncate

Constants

```
const (
    ANSIC       = "Mon Jan _2 15:04:05 2006"
    UnixDate    = "Mon Jan _2 15:04:05 MST 2006"
    RubyDate    = "Mon Jan 02 15:04:05 -0700 2006"
    RFC822      = "02 Jan 06 15:04 MST"
    RFC822Z     = "02 Jan 06 15:04 -0700" // 使用数字表示时区的RFC822
    RFC850      = "Monday, 02-Jan-06 15:04:05 MST"
    RFC1123     = "Mon, 02 Jan 2006 15:04:05 MST"
    RFC1123Z    = "Mon, 02 Jan 2006 15:04:05 -0700" // 使用数字表示时
    RFC3339     = "2006-01-02T15:04:05Z07:00"
    RFC3339Nano = "2006-01-02T15:04:05.999999999Z07:00"
    Kitchen     = "3:04PM"
    // 方便的时间戳
    Stamp      = "Jan _2 15:04:05"
    StampMilli = "Jan _2 15:04:05.000"
    StampMicro = "Jan _2 15:04:05.000000"
    StampNano  = "Jan _2 15:04:05.000000000"
)
```

这些预定义的版式用于Time.Format和Time.Parse函数。用在版式中的参考时间是：

```
Mon Jan 2 15:04:05 MST 2006
```

对应的Unix时间是1136239445。因为MST的时区是GMT-0700，参考时间也可以表示为如下：

```
01/02 03:04:05PM '06 -0700
```

要定义你自己的格式，写下该参考时间应用于你的格式的情况；例子请参见ANSIC、StampMicro或Kitchen等常数的值。该模型是为了演示参考时间的格式化效果，如此一来Format和Parse方法可以将相同的转换规则用于一个普通的时间值。

在格式字符串中，用前置的'0'表示一个可以被可以被数字替换的'0'（如果它后面的数字有两位）；使用下划线表示一个可以被数字替换的空格（如果它后面的数字有两位）；以便兼容Unix定长时间格式。

小数点后跟0到多个'0'，表示秒数的小数部分，输出时会生成和'0'一样多的小数位；小数点后跟0到多个'9'，表示秒数的小数部分，输出时会生成和'9'一样多的小数位但会将拖尾的'0'去掉。（只有）解析时，输入可以在秒字段后面紧跟一个小数部分，即使格式字符串里没有指明该部分。此时，小数点及其后全部的数字都会成为秒的小数部分。

数字表示的时区格式如下：

```
-0700  ±hhmm
-07:00 ±hh:mm
```

将格式字符串中的负号替换为Z会触发ISO 8601行为（当时区是UTC时，输出Z而不是时区偏移量），这样：

```
Z0700  Z or ±hhmm
Z07:00 Z or ±hh:mm
```

type ParseError

```
type ParseError struct {
    Layout     string
    Value      string
    LayoutElem string
    ValueElem  string
    Message    string
}
```

ParseError描述解析时间字符串时出现的错误。

func (*ParseError) Error

```
func (e *ParseError) Error() string
```

Error返回ParseError的字符串表示。

type Weekday

```
type Weekday int
```

Weekday代表一周的某一天。

```
const (
    Sunday Weekday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)
```

func (Weekday) String

```
func (d Weekday) String() string
```

String返回该日（周几）的英文名（"Sunday"、"Monday"，……）

type Month

```
type Month int
```

Month代表一年的某个月。

```
const (
    January Month = 1 + iota
    February
    March
    April
    May
    June
    July
    August
    September
    October
    November
    December
)
```

Example

```
_, month, day := time.Now().Date()
if month == time.November && day == 10 {
    fmt.Println("Happy Go day!")
}
```

func (Month) String

```
func (m Month) String() string
```

String返回月份的英文名（"January"，"February"，……）

type Location

```
type Location struct {
    // 内含隐藏或非导出字段
}
```

Location代表一个（关联到某个时间点的）地点，以及该地点所在的时区。

```
var Local *Location = &localLoc
```

Local代表系统本地，对应本地时区。

```
var UTC *Location = &utcLoc
```

UTC代表通用协调时间，对应零时区。

func LoadLocation

```
func LoadLocation(name string) (*Location, error)
```

LoadLocation返回使用给定的名字创建的Location。

如果name是""或"UTC"，返回UTC；如果name是"Local"，返回Local；否则name应该是IANA时区数据库里有记录的地点名（该数据库记录了地点和对应的时区），如"America/New_York"。

LoadLocation函数需要的时区数据库可能不是所有系统都提供，特别是非Unix系统。此时LoadLocation会查找环境变量ZONEINFO指定目录或解压该变量指定的zip文件（如果有该环境变量）；然后查找Unix系统的惯例时区数据安装位置，最后查找$GOROOT/lib/time/zoneinfo.zip。

func FixedZone

```
func FixedZone(name string, offset int) *Location
```

FixedZone使用给定的地点名name和时间偏移量offset（单位秒）创建并返回一个Location

func (*Location) String

```
func (l *Location) String() string
```

String返回对时区信息的描述，返回值绑定为LoadLocation或FixedZone函数创建l时的name参数。

type Time

```
type Time struct {
    // 内含隐藏或非导出字段
}
```

Time代表一个纳秒精度的时间点。

程序中应使用Time类型值来保存和传递时间，而不能用指针。就是说，表示时间的变量和字段，应为time.Time类型，而不是*time.Time.类型。一个Time类型值可以被多个go程同时使用。时间点可以使用Before、After和Equal方法进行比较。Sub方法让两个时间点相减，生成一个Duration类型值（代表时间段）。Add方法给一个时间点加上一个时间段，生成一个新的Time类型时间点。

Time零值代表时间点January 1, year 1, 00:00:00.000000000 UTC。因为本时间点一般不会出现在使用中，IsZero方法提供了检验时间是否显式初始化的一个简单途径。

每一个时间都具有一个地点信息（及对应地点的时区信息），当计算时间的表示格式时，如Format、Hour和Year等方法，都会考虑该信息。Local、UTC和In方法返回一个指定时区（但指向同一时间点）的Time。修改地点/时区信息只是会改变其表示；不会修改被表示的时间点，因此也不会影响其计算。

func Date

```
func Date(year int, month Month, day, hour, min, sec, nsec int, loc
```

Date返回一个时区为loc、当地时间为：

```
year-month-day hour:min:sec + nsec nanoseconds
```

的时间点。

month、day、hour、min、sec和nsec的值可能会超出它们的正常范围，在转换前函数会自动将之规范化。如October 32被修正为November 1。

夏时制的时区切换会跳过或重复时间。如，在美国，March 13, 2011 2:15am从来不会出现，而November 6, 2011 1:15am 会出现两次。此时，时区的选择和时间是没有良好定义的。Date会返回在时区切换的两个时区其中一个时区

正确的时间，但本函数不会保证在哪一个时区正确。

如果loc为nil会panic。

Example

```
t := time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
fmt.Printf("Go launched at %s\n", t.Local())
```

Output:

```
Go launched at 2009-11-10 15:00:00 -0800 PST
```

func Now

```
func Now() Time
```

Now返回当前本地时间。

func Parse

```
func Parse(layout, value string) (Time, error)
```

Parse解析一个格式化的时间字符串并返回它代表的时间。layout定义了参考时间：

```
Mon Jan 2 15:04:05 -0700 MST 2006
```

在输入格式下的字符串表示，作为输入的格式的示例。同样的格式规则会被用于输入字符串。

预定义的ANSIC、UnixDate、RFC3339和其他版式描述了参考时间的标准或便捷表示。要获得更多参考时间的定义和格式，参见本包的ANSIC和其他版式常量。

value中漏掉的元素会被视为0；如果不能是0，会被视为1。因此，解析"3:04pm"会返回对应时间点：Jan 1, year 0, 15:04:00 UTC的Time（注意因为year为0，该时间在Time零值之前）。年份必须在0000..9999范围内。周几会被检查其语法，但是会被忽略。

如果缺少表示时区的信息，Parse会将时区设置为UTC。

当解析具有时区偏移量的时间字符串时，如果该时区偏移量和本地时区相同，Parse会在返回值中将Location设置为本地和本地时区。否则，它会将Location设置为一个虚构的具有该时区偏移量的值。

当解析具有时区缩写的时间字符串时，如果该时区缩写具有已定义的时间偏移量，会使用该偏移量。如果时区缩写是"UTC"，会将该时间视为UTC时间，不考虑Location。如果时区缩写是未知的，Parse会将Location设置为一个虚构的地点为时区缩写，时间偏移量为0的值。这种做法是为了让一个时间可以在同一版式下不丢失信息的被解析和重新格式化；但字符串表示和具体表示的时间点会因为实际时区偏移量而不同。为了避免这些问题，请使用数字表示的时区偏移量，或者使用ParseInLocation函数。

Example

```
// longForm shows by example how the reference time would be repres
// the desired layout.
const longForm = "Jan 2, 2006 at 3:04pm (MST)"
t, _ := time.Parse(longForm, "Feb 3, 2013 at 7:54pm (PST)")
fmt.Println(t)
// shortForm is another way the reference time would be represented
// in the desired layout; it has no time zone present.
// Note: without explicit zone, returns time in UTC.
const shortForm = "2006-Jan-02"
t, _ = time.Parse(shortForm, "2013-Feb-03")
fmt.Println(t)
```

Output:

```
2013-02-03 19:54:00 -0800 PST
2013-02-03 00:00:00 +0000 UTC
```

func ParseInLocation

```
func ParseInLocation(layout, value string, loc *Location) (Time, er
```

ParseInLocation类似Parse但有两个重要的不同之处。第一，当缺少时区信息时，Parse将时间解释为UTC时间，而ParseInLocation将返回值的Location设置为loc；第二，当时间字符串提供了时区偏移量信息时，Parse会尝试去匹配本地时区，而ParseInLocation会去匹配loc。

Example

```
loc, _ := time.LoadLocation("Europe/Berlin")
const longForm = "Jan 2, 2006 at 3:04pm (MST)"
t, _ := time.ParseInLocation(longForm, "Jul 9, 2012 at 5:02am (CES
fmt.Println(t)
// Note: without explicit zone, returns time in given location.
const shortForm = "2006-Jan-02"
t, _ = time.ParseInLocation(shortForm, "2012-Jul-09", loc)
fmt.Println(t)
```

Output:

```
2012-07-09 05:02:00 +0200 CEST
2012-07-09 00:00:00 +0200 CEST
```

func Unix

```
func Unix(sec int64, nsec int64) Time
```

Unix创建一个本地时间，对应sec和nsec表示的Unix时间（从January 1, 1970 UTC至该时间的秒数和纳秒数）。

nsec的值在[0, 999999999]范围外是合法的。

func (Time) Location

```
func (t Time) Location() *Location
```

Location返回t的地点和时区信息。

func (Time) Zone

```
func (t Time) Zone() (name string, offset int)
```

Zone计算t所在的时区，返回该时区的规范名（如"CET"）和该时区相对于UTC的时间偏移量（单位秒）。

func (Time) IsZero

```
func (t Time) IsZero() bool
```

IsZero报告t是否代表Time零值的时间点，January 1, year 1, 00:00:00 UTC。

func (Time) Local

```
func (t Time) Local() Time
```

Local返回采用本地和本地时区，但指向同一时间点的Time。

func (Time) UTC

```
func (t Time) UTC() Time
```

UTC返回采用UTC和零时区，但指向同一时间点的Time。

func (Time) In

```
func (t Time) In(loc *Location) Time
```

In返回采用loc指定的地点和时区，但指向同一时间点的Time。如果loc为nil会panic。

func (Time) Unix

```
func (t Time) Unix() int64
```

Unix将t表示为Unix时间，即从时间点January 1, 1970 UTC到时间点t所经过的时间（单位秒）。

func (Time) UnixNano

```
func (t Time) UnixNano() int64
```

UnixNano将t表示为Unix时间，即从时间点January 1, 1970 UTC到时间点t所经过的时间（单位纳秒）。如果纳秒为单位的unix时间超出了int64能表示的范围，结果是未定义的。注意这就意味着Time零值调用UnixNano方法的话，结果是未定义的。

func (Time) Equal

```
func (t Time) Equal(u Time) bool
```

判断两个时间是否相同，会考虑时区的影响，因此不同时区标准的时间也可以正确比较。本方法和用t==u不同，这种方法还会比较地点和时区信息。

func (Time) Before

```
func (t Time) Before(u Time) bool
```

如果t代表的时间点在u之前，返回真；否则返回假。

func (Time) After

```
func (t Time) After(u Time) bool
```

如果t代表的时间点在u之后，返回真；否则返回假。

func (Time) Date

```
func (t Time) Date() (year int, month Month, day int)
```

返回时间点t对应的年、月、日。

func (Time) Clock

```
func (t Time) Clock() (hour, min, sec int)
```

返回t对应的那一天的时、分、秒。

func (Time) Year

```
func (t Time) Year() int
```

返回时间点t对应的年份。

func (Time) Month

```
func (t Time) Month() Month
```

返回时间点t对应那一年的第几月。

func (Time) ISOWeek

```
func (t Time) ISOWeek() (year, week int)
```

返回时间点t对应的ISO 9601标准下的年份和星期编号。星期编号范围[1,53]，1月1号到1月3号可能属于上一年的最后一周，12月29号到12月31号可能属于下一年的第一周。

func (Time) YearDay

```
func (t Time) YearDay() int
```

返回时间点t对应的那一年的第几天，平年的返回值范围[1,365]，闰年[1,366]。

func (Time) Day

```
func (t Time) Day() int
```

返回时间点t对应那一月的第几日。

func (Time) Weekday

```
func (t Time) Weekday() Weekday
```

返回时间点t对应的那一周的周几。

func (Time) Hour

```
func (t Time) Hour() int
```

返回t对应的那一天的第几小时，范围[0, 23]。

func (Time) Minute

```
func (t Time) Minute() int
```

返回t对应的那一小时的第几分种，范围[0, 59]。

func (Time) Second

```
func (t Time) Second() int
```

返回t对应的那一分钟的第几秒，范围[0, 59]。

func (Time) Nanosecond

```
func (t Time) Nanosecond() int
```

返回t对应的那一秒内的纳秒偏移量，范围[0, 999999999]。

func (Time) Add

```
func (t Time) Add(d Duration) Time
```

Add返回时间点t+d。

func (Time) AddDate

```
func (t Time) AddDate(years int, months int, days int) Time
```

AddDate返回增加了给出的年份、月份和天数的时间点Time。例如，时间点January 1, 2011调用AddDate(-1, 2, 3)会返回March 4, 2010。

AddDate会将结果规范化，类似Date函数的做法。因此，举个例子，给时间点October 31添加一个月，会生成时间点December 1。（从时间点November 31规范化而来）

func (Time) Sub

```
func (t Time) Sub(u Time) Duration
```

返回一个时间段t-u。如果结果超出了Duration可以表示的最大值/最小值，将返回最大值/最小值。要获取时间点t-d（d为Duration），可以使用t.Add(-d)。

func (Time) Round

```
func (t Time) Round(d Duration) Time
```

返回距离t最近的时间点，该时间点应该满足从Time零值到该时间点的时间段能整除d；如果有两个满足要求的时间点，距离t相同，会向上舍入；如果d <= 0，会返回t的拷贝。

Example

```
t := time.Date(0, 0, 0, 12, 15, 30, 918273645, time.UTC)
round := []time.Duration{
    time.Nanosecond,
    time.Microsecond,
    time.Millisecond,
    time.Second,
    2 * time.Second,
    time.Minute,
    10 * time.Minute,
    time.Hour,
}
for _, d := range round {
    fmt.Printf("t.Round(%6s) = %s\n", d, t.Round(d).Format("15:04:0
}
```

Output:

```
t.Round(   1ns) = 12:15:30.918273645
t.Round(   1us) = 12:15:30.918274
t.Round(   1ms) = 12:15:30.918
t.Round(    1s) = 12:15:31
t.Round(    2s) = 12:15:30
t.Round(  1m0s) = 12:16:00
t.Round( 10m0s) = 12:20:00
t.Round(1h0m0s) = 12:00:00
```

func (Time) Truncate

```
func (t Time) Truncate(d Duration) Time
```

类似Round，但是返回的是最接近但早于t的时间点；如果d <= 0，会返回t的拷贝。

Example

```
t, _ := time.Parse("2006 Jan 02 15:04:05", "2012 Dec 07 12:15:30.91
trunc := []time.Duration{
    time.Nanosecond,
    time.Microsecond,
    time.Millisecond,
    time.Second,
    2 * time.Second,
    time.Minute,
    10 * time.Minute,
    time.Hour,
}
for _, d := range trunc {
    fmt.Printf("t.Truncate(%6s) = %s\n", d, t.Truncate(d).Format("1
}
```

Output:

```
t.Truncate(   1ns) = 12:15:30.918273645
t.Truncate(   1us) = 12:15:30.918273
t.Truncate(   1ms) = 12:15:30.918
t.Truncate(    1s) = 12:15:30
t.Truncate(    2s) = 12:15:30
t.Truncate(  1m0s) = 12:15:00
t.Truncate( 10m0s) = 12:10:00
t.Truncate(1h0m0s) = 12:00:00
```

func (Time) Format

```
func (t Time) Format(layout string) string
```

Format根据layout指定的格式返回t代表的时间点的格式化文本表示。layout定义了参考时间：

```
Mon Jan 2 15:04:05 -0700 MST 2006
```

格式化后的字符串表示，它作为期望输出的例子。同样的格式规则会被用于格式化时间。

预定义的ANSIC、UnixDate、RFC3339和其他版式描述了参考时间的标准或便捷表示。要获得更多参考时间的定义和格式，参见本包的ANSIC和其他版式常量。

Example

```
// layout shows by example how the reference time should be represe
const layout = "Jan 2, 2006 at 3:04pm (MST)"
t := time.Date(2009, time.November, 10, 15, 0, 0, 0, time.Local)
fmt.Println(t.Format(layout))
fmt.Println(t.UTC().Format(layout))
```

Output:

```
Nov 10, 2009 at 3:00pm (PST)
Nov 10, 2009 at 11:00pm (UTC)
```

func (Time) String

```
func (t Time) String() string
```

String返回采用如下格式字符串的格式化时间。

```
"2006-01-02 15:04:05.999999999 -0700 MST"
```

func (Time) GobEncode

```
func (t Time) GobEncode() ([]byte, error)
```

GobEncode实现了gob.GobEncoder接口。

func (*Time) GobDecode

```
func (t *Time) GobDecode(data []byte) error
```

GobEncode实现了gob.GobDecoder接口。

func (Time) MarshalBinary

```
func (t Time) MarshalBinary() ([]byte, error)
```

MarshalBinary实现了encoding.BinaryMarshaler接口。

func (*Time) UnmarshalBinary

```
func (t *Time) UnmarshalBinary(data []byte) error
```

UnmarshalBinary实现了encoding.BinaryUnmarshaler接口。

func (Time) MarshalJSON

```
func (t Time) MarshalJSON() ([]byte, error)
```

MarshalJSON实现了json.Marshaler接口。返回值是用双引号括起来的采用RFC 3339格式进行格式化的时间表示，如果需要会提供小于秒的精度。

func (*Time) UnmarshalJSON

```
func (t *Time) UnmarshalJSON(data []byte) (err error)
```

UnmarshalJSON实现了json.Unmarshaler接口。时间被期望是双引号括起来的RFC 3339格式。

func (Time) MarshalText

```
func (t Time) MarshalText() ([]byte, error)
```

MarshalText实现了encoding.TextMarshaler接口。返回值是采用RFC 3339格式进行格式化的时间表示，如果需要会提供小于秒的精度。

func (*Time) UnmarshalText

```
func (t *Time) UnmarshalText(data []byte) (err error)
```

UnmarshalText实现了encoding.TextUnmarshaler接口。时间被期望采用RFC 3339格式。

type Duration

```
type Duration int64
```

Duration类型代表两个时间点之间经过的时间，以纳秒为单位。可表示的最长时间段大约290年。

```
const (
    Nanosecond  Duration = 1
    Microsecond          = 1000 * Nanosecond
    Millisecond          = 1000 * Microsecond
    Second               = 1000 * Millisecond
    Minute               = 60 * Second
    Hour                 = 60 * Minute
)
```

常用的时间段。没有定义一天或超过一天的单元，以避免夏时制的时区切换的混乱。

要将Duration类型值表示为某时间单元的个数，用除法：

```
second := time.Second
fmt.Print(int64(second/time.Millisecond)) // prints 1000
```

要将整数个某时间单元表示为Duration类型值，用乘法：

```
seconds := 10
fmt.Print(time.Duration(seconds)*time.Second) // prints 10s
```

Example

```
t0 := time.Now()
expensiveCall()
t1 := time.Now()
fmt.Printf("The call took %v to run.\n", t1.Sub(t0))
```

func ParseDuration

```
func ParseDuration(s string) (Duration, error)
```

ParseDuration解析一个时间段字符串。一个时间段字符串是一个序列，每个片段包含可选的正负号、十进制数、可选的小数部分和单位后缀，如"300ms"、"-1.5h"、"2h45m"。合法的单位有"ns"、"us" /"µs"、"ms"、"s"、"m"、"h"。

func Since

```
func Since(t Time) Duration
```

Since返回从t到现在经过的时间，等价于time.Now().Sub(t)。

func (Duration) Hours

```
func (d Duration) Hours() float64
```

Hours将时间段表示为float64类型的小时数。

func (Duration) Minutes

```
func (d Duration) Minutes() float64
```

Hours将时间段表示为float64类型的分钟数。

func (Duration) Seconds

```
func (d Duration) Seconds() float64
```

Hours将时间段表示为float64类型的秒数。

func (Duration) Nanoseconds

```
func (d Duration) Nanoseconds() int64
```

Hours将时间段表示为int64类型的纳秒数，等价于int64(d)。

func (Duration) String

```
func (d Duration) String() string
```

返回时间段采用"72h3m0.5s"格式的字符串表示。最前面可以有符号，数字+单位为一个单元，开始部分的0值单元会被省略；如果时间段<1s，会使用"ms"、"us"、"ns"来保证第一个单元的数字不是0；如果时间段为0，会返回"0"。

type Timer

```
type Timer struct {
    C <-chan Time
    // 内含隐藏或非导出字段
}
```

Timer类型代表单次时间事件。当Timer到期时，当时的时间会被发送给C，除非Timer是被AfterFunc函数创建的。

func NewTimer

```
func NewTimer(d Duration) *Timer
```

NewTimer创建一个Timer，它会在最少过去时间段d后到期，向其自身的C字段发送当时的时间。

func AfterFunc

```
func AfterFunc(d Duration, f func()) *Timer
```

AfterFunc另起一个go程等待时间段d过去，然后调用f。它返回一个Timer，可以通过调用其Stop方法来取消等待和对f的调用。

func (*Timer) Reset

```
func (t *Timer) Reset(d Duration) bool
```

Reset使t重新开始计时，（本方法返回后再）等待时间段d过去后到期。如果调用时t还在等待中会返回真；如果t已经到期或者被停止了会返回假。

func (*Timer) Stop

```
func (t *Timer) Stop() bool
```

Stop停止Timer的执行。如果停止了t会返回真；如果t已经被停止或者过期了会返回假。Stop不会关闭通道t.C，以避免从该通道的读取不正确的成功。

type Ticker

```
type Ticker struct {
    C <-chan Time // 周期性传递时间信息的通道
    // 内含隐藏或非导出字段
}
```

Ticker保管一个通道，并每隔一段时间向其传递"tick"。

func NewTicker

```
func NewTicker(d Duration) *Ticker
```

NewTicker返回一个新的Ticker，该Ticker包含一个通道字段，并会每隔时间段d就向该通道发送当时的时间。它会调整时间间隔或者丢弃tick信息以适应反应慢的接收者。如果d<=0会panic。关闭该Ticker可以释放相关资源。

func (*Ticker) Stop

```
func (t *Ticker) Stop()
```

Stop关闭一个Ticker。在关闭后，将不会发送更多的tick信息。Stop不会关闭通道t.C，以避免从该通道的读取不正确的成功。

func Sleep

```
func Sleep(d Duration)
```

Sleep阻塞当前go程至少d代表的时间段。d<=0时，Sleep会立刻返回。

Example

```
time.Sleep(100 * time.Millisecond)
```

func After

```
func After(d Duration) <-chan Time
```

After会在另一线程经过时间段d后向返回值发送当时的时间。等价于NewTimer(d).C。

Example

```
select {
case m := <-c:
    handle(m)
case <-time.After(5 * time.Minute):
    fmt.Println("timed out")
}
```

func Tick

```
func Tick(d Duration) <-chan Time
```

Tick是NewTicker的封装，只提供对Ticker的通道的访问。如果不需要关闭Ticker，本函数就很方便。

Example

```
c := time.Tick(1 * time.Minute)
for now := range c {
    fmt.Printf("%v %s\n", now, statusUpdate())
}
```

package unicode

```
import "unicode"
```

Package unicode provides data and functions to test some properties of Unicode code points.

Index

Constants

```
const (
    MaxRune         = '\U0010FFFF' // 最大的合法unicode码值
    ReplacementChar = '\uFFFD'     // 表示不合法的unicode码值
    MaxASCII        = '\u007F'     // 最大的ASCII值
    MaxLatin1       = '\u00FF'     // 最大的Latin-1值
)
```

```
const (
    UpperCase = iota
    LowerCase
    TitleCase
    MaxCase
)
```

下值可用于CaseRange类型里的数组类型Delta字段，用于码值映射。

```
const (
    UpperLower = MaxRune + 1 // 不能是合法的delta值
)
```

如果一个CaseRange类型的的Delta字段使用了UpperLower，就表示该CaseRange表示的序列格式为：Upper Lower Upper Lower

```
const Version = "6.3.0"
```

Version是本包采用的unicode版本（以及Variables栏里那些乱七八糟的字符集的来源）。

Variables

```
var (
    Cc     = _Cc // Cc is the set of Unicode characters in category
    Cf     = _Cf // Cf is the set of Unicode characters in category
    Co     = _Co // Co is the set of Unicode characters in category
    Cs     = _Cs // Cs is the set of Unicode characters in category
    Digit  = _Nd // Digit is the set of Unicode characters with the
    Nd     = _Nd // Nd is the set of Unicode characters in category
    Letter = _L  // Letter/L is the set of Unicode letters, categor
    L      = _L
    Lm     = _Lm // Lm is the set of Unicode characters in category
    Lo     = _Lo // Lo is the set of Unicode characters in category
    Lower  = _Ll // Lower is the set of Unicode lower case letters.
    Ll     = _Ll // Ll is the set of Unicode characters in category
    Mark   = _M  // Mark/M is the set of Unicode mark characters, c
    M      = _M
    Mc     = _Mc // Mc is the set of Unicode characters in category
    Me     = _Me // Me is the set of Unicode characters in category
    Mn     = _Mn // Mn is the set of Unicode characters in category
    Nl     = _Nl // Nl is the set of Unicode characters in category
    No     = _No // No is the set of Unicode characters in category
    Number = _N  // Number/N is the set of Unicode number character
    N      = _N
    Other  = _C // Other/C is the set of Unicode control and specia
    C      = _C
    Pc     = _Pc // Pc is the set of Unicode characters in category
    Pd     = _Pd // Pd is the set of Unicode characters in category
    Pe     = _Pe // Pe is the set of Unicode characters in category
    Pf     = _Pf // Pf is the set of Unicode characters in category
    Pi     = _Pi // Pi is the set of Unicode characters in category
    Po     = _Po // Po is the set of Unicode characters in category
    Ps     = _Ps // Ps is the set of Unicode characters in category
    Punct  = _P  // Punct/P is the set of Unicode punctuation chara
    P      = _P
    Sc     = _Sc // Sc is the set of Unicode characters in category
    Sk     = _Sk // Sk is the set of Unicode characters in category
    Sm     = _Sm // Sm is the set of Unicode characters in category
    So     = _So // So is the set of Unicode characters in category
    Space  = _Z  // Space/Z is the set of Unicode space characters,
    Z      = _Z
    Symbol = _S // Symbol/S is the set of Unicode symbol characters
    S      = _S
    Title  = _Lt // Title is the set of Unicode title case letters.
    Lt     = _Lt // Lt is the set of Unicode characters in category
    Upper  = _Lu // Upper is the set of Unicode upper case letters.
    Lu     = _Lu // Lu is the set of Unicode characters in category
    Zl     = _Zl // Zl is the set of Unicode characters in category
    Zp     = _Zp // Zp is the set of Unicode characters in category
    Zs     = _Zs // Zs is the set of Unicode characters in category
)
```

这些变量的类型是*RangeTable。

```
var (
    Arabic                 = _Arabic                 // Arabic is t
    Armenian               = _Armenian               // Armenian is
    Avestan                = _Avestan                // Avestan is
    Balinese               = _Balinese               // Balinese is
    Bamum                  = _Bamum                  // Bamum is th
    Batak                  = _Batak                  // Batak is th
    Bengali                = _Bengali                // Bengali is
    Bopomofo               = _Bopomofo               // Bopomofo is
    Brahmi                 = _Brahmi                 // Brahmi is t
    Braille                = _Braille                // Braille is
    Buginese               = _Buginese               // Buginese is
    Buhid                  = _Buhid                  // Buhid is th
    Canadian_Aboriginal    = _Canadian_Aboriginal    // Canadian_Ab
    Carian                 = _Carian                 // Carian is t
    Chakma                 = _Chakma                 // Chakma is t
    Cham                   = _Cham                   // Cham is the
    Cherokee               = _Cherokee               // Cherokee is
    Common                 = _Common                 // Common is t
    Coptic                 = _Coptic                 // Coptic is t
    Cuneiform              = _Cuneiform              // Cuneiform i
    Cypriot                = _Cypriot                // Cypriot is
    Cyrillic               = _Cyrillic               // Cyrillic is
    Deseret                = _Deseret                // Deseret is
    Devanagari             = _Devanagari             // Devanagari
    Egyptian_Hieroglyphs   = _Egyptian_Hieroglyphs   // Egyptian_Hi
    Ethiopic               = _Ethiopic               // Ethiopic is
    Georgian               = _Georgian               // Georgian is
    Glagolitic             = _Glagolitic             // Glagolitic
    Gothic                 = _Gothic                 // Gothic is t
    Greek                  = _Greek                  // Greek is th
    Gujarati               = _Gujarati               // Gujarati is
    Gurmukhi               = _Gurmukhi               // Gurmukhi is
    Han                    = _Han                    // Han is the
    Hangul                 = _Hangul                 // Hangul is t
    Hanunoo                = _Hanunoo                // Hanunoo is
    Hebrew                 = _Hebrew                 // Hebrew is t
    Hiragana               = _Hiragana               // Hiragana is
    Imperial_Aramaic       = _Imperial_Aramaic       // Imperial_Ar
    Inherited              = _Inherited              // Inherited i
    Inscriptional_Pahlavi  = _Inscriptional_Pahlavi  // Inscription
    Inscriptional_Parthian = _Inscriptional_Parthian // Inscription
    Javanese               = _Javanese               // Javanese is
    Kaithi                 = _Kaithi                 // Kaithi is t
    Kannada                = _Kannada                // Kannada is
    Katakana               = _Katakana               // Katakana is
    Kayah_Li               = _Kayah_Li               // Kayah_Li is
    Kharoshthi             = _Kharoshthi             // Kharoshthi
    Khmer                  = _Khmer                  // Khmer is th
    Lao                    = _Lao                    // Lao is the
```

```
    Latin                  = _Latin                  // Latin is th
    Lepcha                 = _Lepcha                 // Lepcha is t
    Limbu                  = _Limbu                  // Limbu is th
    Linear_B               = _Linear_B               // Linear_B is
    Lisu                   = _Lisu                   // Lisu is the
    Lycian                 = _Lycian                 // Lycian is t
    Lydian                 = _Lydian                 // Lydian is t
    Malayalam              = _Malayalam              // Malayalam i
    Mandaic                = _Mandaic                // Mandaic is
    Meetei_Mayek           = _Meetei_Mayek           // Meetei_Maye
    Meroitic_Cursive       = _Meroitic_Cursive       // Meroitic_Cu
    Meroitic_Hieroglyphs   = _Meroitic_Hieroglyphs   // Meroitic_Hi
    Miao                   = _Miao                   // Miao is the
    Mongolian              = _Mongolian              // Mongolian i
    Myanmar                = _Myanmar                // Myanmar is
    New_Tai_Lue            = _New_Tai_Lue            // New_Tai_Lue
    Nko                    = _Nko                    // Nko is the
    Ogham                  = _Ogham                  // Ogham is th
    Ol_Chiki               = _Ol_Chiki               // Ol_Chiki is
    Old_Italic             = _Old_Italic             // Old_Italic
    Old_Persian            = _Old_Persian            // Old_Persian
    Old_South_Arabian      = _Old_South_Arabian      // Old_South_A
    Old_Turkic             = _Old_Turkic             // Old_Turkic
    Oriya                  = _Oriya                  // Oriya is th
    Osmanya                = _Osmanya                // Osmanya is
    Phags_Pa               = _Phags_Pa               // Phags_Pa is
    Phoenician             = _Phoenician             // Phoenician
    Rejang                 = _Rejang                 // Rejang is t
    Runic                  = _Runic                  // Runic is th
    Samaritan              = _Samaritan              // Samaritan i
    Saurashtra             = _Saurashtra             // Saurashtra
    Sharada                = _Sharada                // Sharada is
    Shavian                = _Shavian                // Shavian is
    Sinhala                = _Sinhala                // Sinhala is
    Sora_Sompeng           = _Sora_Sompeng           // Sora_Sompen
    Sundanese              = _Sundanese              // Sundanese i
    Syloti_Nagri           = _Syloti_Nagri           // Syloti_Nagr
    Syriac                 = _Syriac                 // Syriac is t
    Tagalog                = _Tagalog                // Tagalog is
    Tagbanwa               = _Tagbanwa               // Tagbanwa is
    Tai_Le                 = _Tai_Le                 // Tai_Le is t
    Tai_Tham               = _Tai_Tham               // Tai_Tham is
    Tai_Viet               = _Tai_Viet               // Tai_Viet is
    Takri                  = _Takri                  // Takri is th
    Tamil                  = _Tamil                  // Tamil is th
    Telugu                 = _Telugu                 // Telugu is t
    Thaana                 = _Thaana                 // Thaana is t
    Thai                   = _Thai                   // Thai is the
    Tibetan                = _Tibetan                // Tibetan is
    Tifinagh               = _Tifinagh               // Tifinagh is
    Ugaritic               = _Ugaritic               // Ugaritic is
    Vai                    = _Vai                    // Vai is the
    Yi                     = _Yi                     // Yi is the s
```

```
)
```

这些变量的类型也是*RangeTable。

```
var (
    ASCII_Hex_Digit                    = _ASCII_Hex_Digit
    Bidi_Control                       = _Bidi_Control
    Dash                               = _Dash
    Deprecated                         = _Deprecated
    Diacritic                          = _Diacritic
    Extender                           = _Extender
    Hex_Digit                          = _Hex_Digit
    Hyphen                             = _Hyphen
    IDS_Binary_Operator                = _IDS_Binary_Operator
    IDS_Trinary_Operator               = _IDS_Trinary_Operator
    Ideographic                        = _Ideographic
    Join_Control                       = _Join_Control
    Logical_Order_Exception            = _Logical_Order_Exception
    Noncharacter_Code_Point            = _Noncharacter_Code_Point
    Other_Alphabetic                   = _Other_Alphabetic
    Other_Default_Ignorable_Code_Point = _Other_Default_Ignorable_(
    Other_Grapheme_Extend              = _Other_Grapheme_Extend
    Other_ID_Continue                  = _Other_ID_Continue
    Other_ID_Start                     = _Other_ID_Start
    Other_Lowercase                    = _Other_Lowercase
    Other_Math                         = _Other_Math
    Other_Uppercase                    = _Other_Uppercase
    Pattern_Syntax                     = _Pattern_Syntax
    Pattern_White_Space                = _Pattern_White_Space
    Quotation_Mark                     = _Quotation_Mark
    Radical                            = _Radical
    STerm                              = _STerm
    Soft_Dotted                        = _Soft_Dotted
    Terminal_Punctuation               = _Terminal_Punctuation
    Unified_Ideograph                  = _Unified_Ideograph
    Variation_Selector                 = _Variation_Selector
    White_Space                        = _White_Space
)
```

这些变量的类型还是*RangeTable。

```
var CaseRanges = _CaseRanges
```

CaseRanges is the table describing case mappings for all letters with non-self mappings.

```
var Categories = map[string]*RangeTable{
    "C":  C,
    "Cc": Cc,
    "Cf": Cf,
    "Co": Co,
    "Cs": Cs,
    "L":  L,
    "Ll": Ll,
    "Lm": Lm,
    "Lo": Lo,
    "Lt": Lt,
    "Lu": Lu,
    "M":  M,
    "Mc": Mc,
    "Me": Me,
    "Mn": Mn,
    "N":  N,
    "Nd": Nd,
    "Nl": Nl,
    "No": No,
    "P":  P,
    "Pc": Pc,
    "Pd": Pd,
    "Pe": Pe,
    "Pf": Pf,
    "Pi": Pi,
    "Po": Po,
    "Ps": Ps,
    "S":  S,
    "Sc": Sc,
    "Sk": Sk,
    "Sm": Sm,
    "So": So,
    "Z":  Z,
    "Zl": Zl,
    "Zp": Zp,
    "Zs": Zs,
}
```

Categories is the set of Unicode category tables.

```
var FoldCategory = map[string]*RangeTable{
    "Common":    foldCommon,
    "Greek":     foldGreek,
    "Inherited": foldInherited,
    "L":         foldL,
    "Ll":        foldLl,
    "Lt":        foldLt,
    "Lu":        foldLu,
    "M":         foldM,
    "Mn":        foldMn,
}
```

FoldCategory maps a category name to a table of code points outside the category that are equivalent under simple case folding to code points inside the category. If there is no entry for a category name, there are no such points.

```
var FoldScript = map[string]*RangeTable{}
```

FoldScript maps a script name to a table of code points outside the script that are equivalent under simple case folding to code points inside the script. If there is no entry for a script name, there are no such points.

```
var GraphicRanges = []*RangeTable{
    L, M, N, P, S, Zs,
}
```

GraphicRanges defines the set of graphic characters according to Unicode.

```
var PrintRanges = []*RangeTable{
    L, M, N, P, S,
}
```

PrintRanges defines the set of printable characters according to Go. ASCII space, U+0020, is handled separately.

```
var Properties = map[string]*RangeTable{
    "ASCII_Hex_Digit":                    ASCII_Hex_Digit,
    "Bidi_Control":                       Bidi_Control,
    "Dash":                               Dash,
    "Deprecated":                         Deprecated,
    "Diacritic":                          Diacritic,
    "Extender":                           Extender,
    "Hex_Digit":                          Hex_Digit,
    "Hyphen":                             Hyphen,
    "IDS_Binary_Operator":                IDS_Binary_Operator,
    "IDS_Trinary_Operator":               IDS_Trinary_Operator,
    "Ideographic":                        Ideographic,
    "Join_Control":                       Join_Control,
    "Logical_Order_Exception":            Logical_Order_Exception,
    "Noncharacter_Code_Point":            Noncharacter_Code_Point,
    "Other_Alphabetic":                   Other_Alphabetic,
    "Other_Default_Ignorable_Code_Point": Other_Default_Ignorable_C
    "Other_Grapheme_Extend":              Other_Grapheme_Extend,
    "Other_ID_Continue":                  Other_ID_Continue,
    "Other_ID_Start":                     Other_ID_Start,
    "Other_Lowercase":                    Other_Lowercase,
    "Other_Math":                         Other_Math,
    "Other_Uppercase":                    Other_Uppercase,
    "Pattern_Syntax":                     Pattern_Syntax,
    "Pattern_White_Space":                Pattern_White_Space,
    "Quotation_Mark":                     Quotation_Mark,
    "Radical":                            Radical,
    "STerm":                              STerm,
    "Soft_Dotted":                        Soft_Dotted,
    "Terminal_Punctuation":               Terminal_Punctuation,
    "Unified_Ideograph":                  Unified_Ideograph,
    "Variation_Selector":                 Variation_Selector,
    "White_Space":                        White_Space,
}
```

Properties is the set of Unicode property tables.

```
var Scripts = map[string]*RangeTable{
    "Arabic":                 Arabic,
    "Armenian":               Armenian,
    "Avestan":                Avestan,
    "Balinese":               Balinese,
    "Bamum":                  Bamum,
    "Batak":                  Batak,
    "Bengali":                Bengali,
    "Bopomofo":               Bopomofo,
    "Brahmi":                 Brahmi,
    "Braille":                Braille,
    "Buginese":               Buginese,
    "Buhid":                  Buhid,
```

```
    "Canadian_Aboriginal":    Canadian_Aboriginal,
    "Carian":                 Carian,
    "Chakma":                 Chakma,
    "Cham":                   Cham,
    "Cherokee":               Cherokee,
    "Common":                 Common,
    "Coptic":                 Coptic,
    "Cuneiform":              Cuneiform,
    "Cypriot":                Cypriot,
    "Cyrillic":               Cyrillic,
    "Deseret":                Deseret,
    "Devanagari":             Devanagari,
    "Egyptian_Hieroglyphs":   Egyptian_Hieroglyphs,
    "Ethiopic":               Ethiopic,
    "Georgian":               Georgian,
    "Glagolitic":             Glagolitic,
    "Gothic":                 Gothic,
    "Greek":                  Greek,
    "Gujarati":               Gujarati,
    "Gurmukhi":               Gurmukhi,
    "Han":                    Han,
    "Hangul":                 Hangul,
    "Hanunoo":                Hanunoo,
    "Hebrew":                 Hebrew,
    "Hiragana":               Hiragana,
    "Imperial_Aramaic":       Imperial_Aramaic,
    "Inherited":              Inherited,
    "Inscriptional_Pahlavi":  Inscriptional_Pahlavi,
    "Inscriptional_Parthian": Inscriptional_Parthian,
    "Javanese":               Javanese,
    "Kaithi":                 Kaithi,
    "Kannada":                Kannada,
    "Katakana":               Katakana,
    "Kayah_Li":               Kayah_Li,
    "Kharoshthi":             Kharoshthi,
    "Khmer":                  Khmer,
    "Lao":                    Lao,
    "Latin":                  Latin,
    "Lepcha":                 Lepcha,
    "Limbu":                  Limbu,
    "Linear_B":               Linear_B,
    "Lisu":                   Lisu,
    "Lycian":                 Lycian,
    "Lydian":                 Lydian,
    "Malayalam":              Malayalam,
    "Mandaic":                Mandaic,
    "Meetei_Mayek":           Meetei_Mayek,
    "Meroitic_Cursive":       Meroitic_Cursive,
    "Meroitic_Hieroglyphs":   Meroitic_Hieroglyphs,
    "Miao":                   Miao,
    "Mongolian":              Mongolian,
    "Myanmar":                Myanmar,
    "New_Tai_Lue":            New_Tai_Lue,
```

```
    "Nko":                     Nko,
    "Ogham":                   Ogham,
    "Ol_Chiki":                Ol_Chiki,
    "Old_Italic":              Old_Italic,
    "Old_Persian":             Old_Persian,
    "Old_South_Arabian":       Old_South_Arabian,
    "Old_Turkic":              Old_Turkic,
    "Oriya":                   Oriya,
    "Osmanya":                 Osmanya,
    "Phags_Pa":                Phags_Pa,
    "Phoenician":              Phoenician,
    "Rejang":                  Rejang,
    "Runic":                   Runic,
    "Samaritan":               Samaritan,
    "Saurashtra":              Saurashtra,
    "Sharada":                 Sharada,
    "Shavian":                 Shavian,
    "Sinhala":                 Sinhala,
    "Sora_Sompeng":            Sora_Sompeng,
    "Sundanese":               Sundanese,
    "Syloti_Nagri":            Syloti_Nagri,
    "Syriac":                  Syriac,
    "Tagalog":                 Tagalog,
    "Tagbanwa":                Tagbanwa,
    "Tai_Le":                  Tai_Le,
    "Tai_Tham":                Tai_Tham,
    "Tai_Viet":                Tai_Viet,
    "Takri":                   Takri,
    "Tamil":                   Tamil,
    "Telugu":                  Telugu,
    "Thaana":                  Thaana,
    "Thai":                    Thai,
    "Tibetan":                 Tibetan,
    "Tifinagh":                Tifinagh,
    "Ugaritic":                Ugaritic,
    "Vai":                     Vai,
    "Yi":                      Yi,
}
```

Scripts is the set of Unicode script tables.

type CaseRange

```
type CaseRange struct {
    Lo    uint32
    Hi    uint32
    Delta d // d为[MaxCase]rune的命名类型
}
```

代表简单的unicode码值的一一映射。范围为[Lo, Hi]，步长为1。

该范围内的每个值+Delta[UpperCase]表示对应的大写字母；

该范围内的每个值+Delta[LowerCase]表示对应的小写字母；

该范围内的每个值+Delta[TitleCase]表示对应的标题字母。

Delta数组里的值可为负数或零。如果Delta数组是：

```
{UpperLower, UpperLower, UpperLower}
```

表示[Lo, Hi]范围的字符序列是交替的、对应的大写字母小写字母对。否则常数UpperLower不能用Delta数组里。

type Range16

```
type Range16 struct {
    Lo     uint16
    Hi     uint16
    Stride uint16
}
```

代表一系列16位unicode码值，范围为Lo到Hi（可以是Lo/Hi），步长为Stride。

type Range32

```
type Range32 struct {
    Lo     uint32
    Hi     uint32
    Stride uint32
}
```

代表一系列32位unicode码值，范围为Lo到Hi（可以是Lo/Hi），步长为Stride；Lo和Hi必须大于等于1<<16。

type RangeTable

```
type RangeTable struct {
    R16         []Range16
    R32         []Range32  // R32不能包含低于0x10000（即1<<16）的值
    LatinOffset int // R16字段中Hi <= MaxLatin1的成员数
}
```

通过列出集合中码值的范围，定义了一个unicode码值的集合。出于节省空间，范围保存在两个切片里，分别保存16位字符的范围和32位字符的范围。R16和R32必须是有序排列的，且互不重叠的。

type SpecialCase

```
type SpecialCase []CaseRange
```

SpecialCase代表特定语言的字符映射，如土耳其语。本类型的方法（通过覆盖）定制了标准映射。

```
var AzeriCase SpecialCase = _TurkishCase
```

```
var TurkishCase SpecialCase = _TurkishCase
```

func (SpecialCase) ToLower

```
func (special SpecialCase) ToLower(r rune) rune
```

按特定映射，返回对应的小写字母。

func (SpecialCase) ToUpper

```
func (special SpecialCase) ToUpper(r rune) rune
```

按特定映射，返回对应的大写字母。

func (SpecialCase) ToTitle

```
func (special SpecialCase) ToTitle(r rune) rune
```

按特定映射，返回对应的标题字母。

func Is

```
func Is(rangeTab *RangeTable, r rune) bool
```

函数报告r是否在rangeTab指定的字符范围内。

func In

```
func In(r rune, ranges ...*RangeTable) bool
```

函数报告r是否是给出的某个字母集的成员。

func IsOneOf

```
func IsOneOf(ranges []*RangeTable, r rune) bool
```

函数报告r是否是ranges某个成员指定的字符范围内。本函数的功能类似In，应优先使用In函数。

func IsSpace

```
func IsSpace(r rune) bool
```

IsSpace报告一个字符是否是空白字符。在Latin-1字符空间中，空白字符为：

```
'\t', '\n', '\v', '\f', '\r', ' ', U+0085 (NEL), U+00A0 (NBSP).
```

其它的空白字符请参见策略Z和属性Pattern_White_Space。

func IsDigit

```
func IsDigit(r rune) bool
```

IsDigit报告一个字符是否是十进制数字字符。

func IsNumber

```
func IsNumber(r rune) bool
```

IsNumber报告一个字符是否是数字字符，参见策略N。

func IsLetter

```
func IsLetter(r rune) bool
```

IsLetter报告一个字符是否是字母，参见策略L。

func IsGraphic

```
func IsGraphic(r rune) bool
```

报告一个字符是否是unicode图形。包括字母、标记、数字、符号、标点、空白，参见L、M、N、P、S、Zs。

func IsMark

```
func IsMark(r rune) bool
```

IsMark报告一个字符是否是标记字符，参见策略M。

func IsPrint

```
func IsPrint(r rune) bool
```

IsPrint一个字符是否是go的可打印字符。本函数基本和IsGraphic一致，只是ASCII空白字符U+0020会返回假。

func IsControl

```
func IsControl(r rune) bool
```

IsControl报告一个字符是否是控制字符，主要是策略C的字符和一些其他的字符如代理字符。

func IsPunct

```
func IsPunct(r rune) bool
```

IsPunct报告一个字符是否是unicode标点字符，参见策略P。

func IsSymbol

```
func IsSymbol(r rune) bool
```

IsPunct报告一个字符是否是unicode符号字符。

func IsLower

```
func IsLower(r rune) bool
```

返回字符是否是小写字母。

func IsUpper

```
func IsUpper(r rune) bool
```

返回字符是否是大写字母。

func IsTitle

```
func IsTitle(r rune) bool
```

返回字符是否是标题字母。

func To

```
func To(_case int, r rune) rune
```

返回_case指定的对应类型的字母：UpperCase、LowerCase、TitleCase。

func ToLower

```
func ToLower(r rune) rune
```

返回对应的小写字母。

func ToUpper

```
func ToUpper(r rune) rune
```

返回对应的大写字母。

func ToTitle

```
func ToTitle(r rune) rune
```

返回对应的标题字母。

func SimpleFold

```
func SimpleFold(r rune) rune
```

SimpleFold函数迭代在unicode标准字符映射中互相对应的unicode码值。在与r对应的码值中（包括r自身），会返回最小的那个大于r的字符（如果有）；否则返回映射中最小的字符。

举例：

```
SimpleFold('A') = 'a'
SimpleFold('a') = 'A'
SimpleFold('K') = 'k'
SimpleFold('k') = '\u212A' (Kelvin symbol, K)
SimpleFold('\u212A') = 'K'
SimpleFold('1') = '1'
```

Bugs

☞ 没有全字符（包含多个rune的字符）折叠的机制。

package utf16

```
import "unicode/utf16"
```

utf16包实现了UTF-16序列的编解码。

Index

func IsSurrogate

```
func IsSurrogate(r rune) bool
```

返回r是否可以编码为一个utf-16的代理对。

func DecodeRune

```
func DecodeRune(r1, r2 rune) rune
```

将utf-16代理对(r1, r2)解码为unicode码值。如果代理对不合法，会返回U+FFFD。

func Decode

```
func Decode(s []uint16) []rune
```

将utf-16序列解码为unicode码值序列。

func EncodeRune

```
func EncodeRune(r rune) (r1, r2 rune)
```

将unicode码值r编码为一个utf-16的代理对。如果不能编码，会返回(U+FFFD, U+FFFD)。

func Encode

```
func Encode(s []rune) []uint16
```

将unicode码值序列编码为utf-16序列。

package utf8

```
import "unicode/utf8"
```

utf8包实现了对utf-8文本的常用函数和常数的支持，包括rune和utf-8编码byte序列之间互相翻译的函数。

Index

Examples

Constants

```
const (
    RuneError = '\uFFFD'     // 错误的Rune或"Unicode replacement cha
    RuneSelf  = 0x80         // 低于RunSelf的字符用代表单字节的同一值表ラ
    MaxRune   = '\U0010FFFF' // 最大的合法unicode码值
    UTFMax    = 4            // 最大的utf-8编码的unicode字符的长度
)
```

编码的基础常数。

func ValidRune

```
func ValidRune(r rune) bool
```

判断r是否可以编码为合法的utf-8序列。

Example

```
valid := 'a'
invalid := rune(0xfffffff)
fmt.Println(utf8.ValidRune(valid))
fmt.Println(utf8.ValidRune(invalid))
```

Output:

```
true
false
```

func RuneLen

```
func RuneLen(r rune) int
```

返回r编码后的字节数。如果r不是一个合法的可编码为utf-8序列的值，会返回-1。

Example

```
fmt.Println(utf8.RuneLen('a'))
fmt.Println(utf8.RuneLen('界'))
```

Output:

```
1
3
```

func RuneStart

```
func RuneStart(b byte) bool
```

报告字节b是否可以作为某个rune编码后的第一个字节。第二个即之后的字节总是将左端两个字位设为10。

Example

```
buf := []byte("a界")
fmt.Println(utf8.RuneStart(buf[0]))
fmt.Println(utf8.RuneStart(buf[1]))
fmt.Println(utf8.RuneStart(buf[2]))
```

Output:

```
true
true
false
```

func FullRune

```
func FullRune(p []byte) bool
```

报告切片p是否以一个码值的完整utf-8编码开始。不合法的编码因为会被转换为宽度1的错误码值而被视为完整的。

Example

```
buf := []byte{228, 184, 150} // 世
fmt.Println(utf8.FullRune(buf))
fmt.Println(utf8.FullRune(buf[:2]))
```

Output:

```
true
false
```

func FullRuneInString

```
func FullRuneInString(s string) bool
```

函数类似FullRune但输入参数是字符串。

Example

```
str := "世"
fmt.Println(utf8.FullRuneInString(str))
fmt.Println(utf8.FullRuneInString(str[:2]))
```

Output:

```
true
false
```

func RuneCount

```
func RuneCount(p []byte) int
```

返回p中的utf-8编码的码值的个数。错误或者不完整的编码会被视为宽度1字节的单个码值。

Example

```
buf := []byte("Hello, 世界")
fmt.Println("bytes =", len(buf))
fmt.Println("runes =", utf8.RuneCount(buf))
```

Output:

```
bytes = 13
runes = 9
```

func RuneCountInString

```
func RuneCountInString(s string) (n int)
```

函数类似RuneCount但输入参数是一个字符串。

Example

```
str := "Hello, 世界"
fmt.Println("bytes =", len(str))
fmt.Println("runes =", utf8.RuneCountInString(str))
```

Output:

```
bytes = 13
runes = 9
```

func Valid

```
func Valid(p []byte) bool
```

返回切片p是否包含完整且合法的utf-8编码序列。

Example

```
valid := []byte("Hello, 世界")
invalid := []byte{0xff, 0xfe, 0xfd}
fmt.Println(utf8.Valid(valid))
fmt.Println(utf8.Valid(invalid))
```

Output:

```
true
false
```

func ValidString

```
func ValidString(s string) bool
```

报告s是否包含完整且合法的utf-8编码序列。

Example

```
valid := "Hello, 世界"
invalid := string([]byte{0xff, 0xfe, 0xfd})
fmt.Println(utf8.ValidString(valid))
fmt.Println(utf8.ValidString(invalid))
```

Output:

```
true
false
```

func EncodeRune

```
func EncodeRune(p []byte, r rune) int
```

EncodeRune将r的utf-8编码序列写入p（p必须有足够的长度），并返回写入的字节数。

Example

```
r := '世'
buf := make([]byte, 3)
n := utf8.EncodeRune(buf, r)
fmt.Println(buf)
fmt.Println(n)
```

Output:

```
[228 184 150]
3
```

func DecodeRune

```
func DecodeRune(p []byte) (r rune, size int)
```

函数解码p开始位置的第一个utf-8编码的码值，返回该码值和编码的字节数。如果编码不合法，会返回(RuneError, 1)。该返回值在正确的utf-8编码情况下是不可能返回的。

如果一个utf-8编码序列格式不正确，或者编码的码值超出utf-8合法码值的范围，或者不是该码值的最短编码，该编码序列即是不合法的。函数不会执行其他的验证。

Example

```
b := []byte("Hello, 世界")
for len(b) > 0 {
    r, size := utf8.DecodeRune(b)
    fmt.Printf("%c %v\n", r, size)
    b = b[size:]
}
```

Output:

```
H 1
e 1
l 1
l 1
o 1
, 1
  1
世 3
界 3
```

func DecodeRuneInString

```
func DecodeRuneInString(s string) (r rune, size int)
```

函数类似DecodeRune但输入参数是字符串。

Example

```
str := "Hello, 世界"
for len(str) > 0 {
    r, size := utf8.DecodeRuneInString(str)
    fmt.Printf("%c %v\n", r, size)
    str = str[size:]
}
```

Output:

```
H 1
e 1
l 1
l 1
o 1
, 1
  1
世 3
界 3
```

func DecodeLastRune

```
func DecodeLastRune(p []byte) (r rune, size int)
```

函数解码p中最后一个utf-8编码序列，返回该码值和编码序列的长度。

Example

```
b := []byte("Hello, 世界")
for len(b) > 0 {
    r, size := utf8.DecodeLastRune(b)
    fmt.Printf("%c %v\n", r, size)
    b = b[:len(b)-size]
}
```

Output:

```
界 3
世 3
  1
, 1
o 1
l 1
l 1
e 1
H 1
```

func DecodeLastRuneInString

```
func DecodeLastRuneInString(s string) (r rune, size int)
```

函数类似DecodeLastRune但输入参数是字符串。

Example

```
str := "Hello, 世界"
for len(str) > 0 {
    r, size := utf8.DecodeLastRuneInString(str)
    fmt.Printf("%c %v\n", r, size)
    str = str[:len(str)-size]
}
```

Output:

```
界 3
世 3
  1
, 1
o 1
l 1
l 1
e 1
H 1
```

package unsafe

```
import "unsafe"
```

unsafe包提供了一些跳过go语言类型安全限制的操作。

Index

type ArbitraryType

```
type ArbitraryType int
```

ArbitraryType在本文档里表示任意一种类型，但并非一个实际存在与unsafe包的类型。

type Pointer

```
type Pointer *ArbitraryType
```

Pointer类型用于表示任意类型的指针。有4个特殊的只能用于Pointer类型的操作：

```
1) 任意类型的指针可以转换为一个Pointer类型值
2) 一个Pointer类型值可以转换为任意类型的指针
3) 一个uintptr类型值可以转换为一个Pointer类型值
4) 一个Pointer类型值可以转换为一个uintptr类型值
```

因此，Pointer类型允许程序绕过类型系统读写任意内存。使用它时必须谨慎。

func Sizeof

```
func Sizeof(v ArbitraryType) uintptr
```

Sizeof返回类型v本身数据所占用的字节数。返回值是“顶层”的数据占有的字节数。例如，若v是一个切片，它会返回该切片描述符的大小，而非该切片底层引用的内存的大小。

func Alignof

```
func Alignof(v ArbitraryType) uintptr
```

Alignof返回类型v的对齐方式（即类型v在内存中占用的字节数）；若是结构体类型的字段的形式，它会返回字段f在该结构体中的对齐方式。

func Offsetof

```
func Offsetof(v ArbitraryType) uintptr
```

Offsetof返回类型v所代表的结构体字段在结构体中的偏移量，它必须为结构体类型的字段的形式。换句话说，它返回该结构起始处与该字段起始处之间的字节数。